AIX версии 7.2

Общие принципы программирования

AIX версии 7.2

Общие принципы программирования

Примечание

Перед началом работы с этим изданием и описанным в нем продуктом ознакомьтесь с информацией, приведенной в разделе “Примечания” на стр. 825.

Данное издание относится к AIX версии 7.2, а также ко всем последующим выпускам и модификациям, если в соответствующих изданиях не будет оговорено обратное.

Содержание

Об этом документе

В этом наборе разделов разработчики приложений найдут полную информацию о создании приложений для операционной системы AIX®. Его можно использовать как руководство по программированию и источник полезных ресурсов. В ней рассматриваются такие вопросы, как обработка ввода-вывода, функции curses, файловые системы и каталоги, анализаторы lex и yacc, использование логических томов в программах, общие библиотеки, поддержка больших программ, создание пакетов, функция трассировки и Инструмент управления системой (SMIT).

Выделение текста

В данном документе применяются следующие специальные обозначения:

Полужирный	Полужирным шрифтом выделены названия команд, подпрограмм, ключевых слов, файлов, структур, каталогов и прочих объектов, имена которых предопределены системой. Кроме того, полужирным шрифтом выделены названия графических объектов, выбираемых пользователем - кнопок, меток, значков и т.д.
Курсив	Курсивом выделены те параметры, имена или значения которых задаются пользователем.
`Непропорциональный`	Непропорциональным шрифтом выделены конкретные значения, текст, который вы можете увидеть на экране, фрагменты программных кодов, системные сообщения и данные, которые вам будет предложено ввести.

Учет регистра символов в AIX

В операционной системе AIX учитывается регистр символов, т.е. различаются прописные и строчные буквы. Например, с помощью команды **ls** можно просмотреть список файлов. При вводе команды `LS` отобразится сообщение `Команда не найдена`. Аналогично, имена файлов **FILEA**, **FiLea** и **filea** считаются разными, даже если эти файлы расположены в одном каталоге. Во избежание нежелательных последствий всегда проверяйте правильность регистра букв.

ISO 9000

При разработке и производстве данного продукта использовались зарегистрированные системы ISO 9000.

Общие концепции программирования

В этом наборе разделов разработчики приложений найдут полную информацию о создании приложений для операционной системы AIX®. Эту информацию можно использовать как руководство по программированию и источник полезных ресурсов. В ней рассматриваются такие вопросы, как обработка ввода-вывода, функции curses, файловые системы и каталоги, анализаторы lex и yacc, использование логических томов в программах, общие библиотеки, поддержка больших программ, создание пакетов, функция трассировки и Инструмент управления системой (SMIT).

Операционная система AIX поддерживает одиночную спецификацию UNIX версии 3 Open Group (UNIX 03), касающуюся переносимости операционных систем на базе UNIX. Для соответствия этой спецификации было добавлено несколько новых и существующих интерфейсов. Для того чтобы выбрать правильный способ разработки переносимых на UNIX 03 приложений, обратитесь к спецификации Open Group UNIX 03, доступной на сайте UNIX System (http://www.unix.org).

Новое в книге Общие концепции программирования

Ознакомьтесь с обновлениями и дополнениями в группе разделов, посвященных общим концепциям программирования.

Обозначение дополнений и изменений

В этом файле PDF дополнения и изменения могут обозначаться символами вертикальной черты (|) в левом поле.

Октябрь 2017 года

Ниже приведено краткое описание изменений, внесенных в разделы из этой книги:

- Добавлен раздел “Статистика памяти” на стр. 236.
- В следующих разделах обновлена информация о новых администраторах тестов и связанных встроенных переменных:
  - “Встроенные переменные класса” на стр. 211
  - “Запуск ProbeVue” на стр. 273
  - “Функции Vue” на стр. 375

Инструменты и утилиты

В этом разделе приведен обзор инструментов и утилит, предназначенных для разработки программ на языке C.

Существует множество инструментов для разработки программ на языке C. В операционной системе предусмотрены средства поддержки следующих этапов разработки программ:

Приведена информация о функциях, которые могут использоваться в программах на языке C.

- Ввод текста программы
- Проверка текста программы
- Компиляция и компоновка программы
- Функции
- Команды оболочки

Ввод текста программы

Для ввода и сохранения текста программ предназначен построчный редактор **ed**. Кроме того, в системе есть и полноэкранный интерактивный редактор, который называется **vi**.

Проверка текста программы

Ниже перечислены команды, предназначенные для проверки правильности текста программ:

команды	Описание
cb	Располагает исходный текст программы на языке C в структурном виде с отступами различной ширины.
cflow	Генерирует логическую схему выполнения программы на языке C.
cxref	Генерирует список всех внешних ссылок для каждого модуля исходной программы на языке C с указанием того, где определены эти ссылки (если они определены в программе).
lint	Проверяет синтаксис и соответствие типов в исходном тексте программы на языке C. Команда **lint** проверяет программу более тщательно, чем компилятор C, и выдает большое количество сообщений о возможных неполадках.

Компиляция и компоновка программы

Для получения исполняемой программы из исходного кода необходимо обработать исходный файл компилятором и редактором связей (компоновщиком).

Компилятор - это программа, которая считывает текст из файла и преобразует его в последовательность команд на языке, понятном системе. *Редактор связей* соединяет программные модули и определяет способ загрузки полученной программы в память. Система создает окончательный исполняемый код программы в несколько этапов:

1. Если файл содержит исходный код на языке высокого уровня, компилятор преобразует его в объектный код.
2. Если файл содержит код на языке ассемблера, ассемблер преобразует его в объектный код.
3. Редактор связей связывает объектные файлы, созданные на предыдущем этапе, со всеми остальными объектными файлами, указанными в команде компилятора.

В операционной системе AIX можно также создавать программы на языках C^{++}, FORTRAN, COBOL, ассемблер и на других языках.

Модули программы могут быть написаны на разных языках. Программа должна содержать основную функцию main, которая должна отвечать за вызов всех прочих процедур и модулей. Для создания объектного кода и компоновки используется компилятор.

Исправление ошибок в программе

С базовой операционной системой поставляются следующие средства отладки:

- Программа символьной отладки **dbx** позволяет отлаживать программы, написанные на языках C, C^{++}, FORTRAN, COBOL и на языке ассемблера.
- Программа отладки **adb** содержит средства анализа, отладки и исправления исполняемых файлов, а также удобна для анализа файлов данных в более сложных форматах, чем ASCII.
- Отладчик ядра KDB и команда kdb - это средства поиска ошибок в ядре операционной системы. Этот отладчик используется для отладки драйверов устройств и расширений ядра.
- Трассировщик служит для локализации неполадок в системе путем отслеживания указанных системных событий.

Если в программном файле обнаружены синтаксические ошибки или несоответствия имен параметров, то строки, содержащие эти ошибки, можно найти и изменить с помощью текстового редактора или программы

поиска и редактирования строк. К последним относятся такие утилиты, как **grep**, **sed** и **awk**. Для внесения масштабных изменений в программные файлы удобно пользоваться сценариями оболочки, автоматизирующими вызов таких утилит.

Сборка и обслуживание программы

В системе предусмотрены специальные средства для управления изменениями программ и для сборки программ из исходных модулей. Эти средства особенно полезны при работе с многомодульными программами.

- Утилита **make** собирает программу из исходных модулей. Команда **make** очень эффективна, поскольку компилирует только те модули, которые были изменены с момента последней сборки.
- Система контроля исходного кода (SCCS) позволяет одновременно работать с несколькими версиями программы без создания отдельного кода для каждой версии. Применение SCCS экономит память, а кроме того, весьма эффективно при разработке крупных проектов, когда требуется хранить много версий больших программ.

Функции

Функции, предоставляемые системными библиотеками, облегчают разработку сложных или повторяющихся программных фрагментов и позволяют разработчику сосредоточиться на нестандартных вопросах.

Команды оболочки

В программы на языке C можно включать многие команды оболочки. Все команды оболочки, используемые в программе, должны быть доступны во всех системах, в которых эта программа должна выполняться.

С помощью функций **fork** и **exec** можно запускать команды оболочки в качестве отдельных процессов вне программы. Функция **system** также позволяет выполнять в программах команды оболочки, а функция **popen** - пользоваться фильтрами оболочки.

Понятия, связанные с данным:

“Обработка строк с помощью sed” на стр. 585
Программа **sed** работает без вмешательства пользователя, запросившего редактирование.

“Создание лексического анализатора с помощью команды lex” на стр. 499
Команда **lex** помогает создать программу на языке C, которая может получать поток символов и преобразовывать его в действия программы.

“Команда make” на стр. 529
В этом разделе рассказано, как можно упростить процедуру повторной компиляции и компоновки с помощью команды **make**.

“Функции, примеры программ и библиотеки” на стр. 707
В данном разделе объясняется, что такое функции, как их использовать, и где они хранятся.

Библиотека Curses

Библиотека curses содержит функции управления дисплеями, которые могут применяться независимо от типа терминала, к которому подключен конкретный дисплей. Библиотека curses поддерживает управление цветом. В ней нет поддержки многобайтовых символов. Термин "символ" в документации по Curses всегда означает "однобайтовый символ".

В дальнейшем в этом документе библиотека curses будет называться просто *curses*.

Основным объектом в программах, использующих curses, является структура данных с информацией об окне. Эта структура позволяет управлять данными, показанными на дисплее терминала. Можно настроить curses так, что дисплей будет рассматриваться как одно большое окно, а можно создать на дисплее

несколько окон. Эти окна могут иметь различные размеры и перекрывать друг друга. Типичное приложение curses содержит одно основное окно, в котором находится одно вложенное окно.

Каждому окну соответствует своя собственная структура данных. Эта структура содержит данные о параметрах этого окна (размер, расположение на дисплее и т.д.). С помощью структуры данных об окне функции Curses получают данные, необходимые для выполнения команд.

Терминология

Рекомендуем вам ознакомиться со следующими терминами, связанными с curses:

Термин	Определение
текущий символ	Символ, на который в настоящий момент указывает логический курсор.
текущая строка	Строка, в которой находится логический курсор.
curscr	Стандартное виртуальное окно curses. Структура curscr (текущий экран) - это внутреннее представление данных, показанных на дисплее. Изменять curscr нельзя.
display	Физический дисплей рабочей станции.
логический курсор	Определяет расположение курсора в каждом окне. Положение логического курсора каждого окна хранится в структуре данных с информацией об этом окне.
панель	Панель - это окно, которое не ограничено размерами экрана
физический курсор	Курсор дисплея. Этот курсор служит для вывода данных на дисплей рабочей станции. У каждого дисплея существует только один физический курсор.
экран	Окно, занимающее весь дисплей. Синоним stdscr.
stdscr	Стандартное виртуальное окно (стандартный экран) curses, которое представляет весь дисплей целиком.
window	Указатель на структуру данных C, а также ее графическое представление на дисплее. Окно можно считать двумерной структурой, которая в любой момент времени отражает вид дисплея или его части.

Соглашения об именах

Одна функция curses может иметь более одной версии. В функциях Curses с несколькими версиями применяются особые правила присвоения имен, позволяющие идентифицировать каждую версию. В соответствии с этими правилами к стандартному имени функции curses добавляется префикс, указываются необходимые для ее запуска аргументы или действия, выполняемые этой функцией. Для имен различных версий функций curses применяются следующие префиксы:

Префикс	Описание
w	Обозначает функцию, для которой в качестве аргумента должно быть задано окно
p	Обозначает функцию, для которой в качестве аргумента должна быть задана панель
mv	Обозначает функцию перехода к указанным координатам

Если есть несколько версий функции curses, не содержащих ни один из предыдущих префиксов, то будет применяться окно curses по умолчанию - stdscr (стандартное окно). Большинство функций, работающих с окном stdscr - это макрокоманды, определенные в файле **/usr/include/curses.h** с помощью операторов **#define**. При компиляции препроцессор заменяет эти операторы. В результате макрокоманды не включаются в откомпилированный код, в данные трассировки, отладки, а также в исходный код curses.

Если функция curses имеет только одну версию, то она не обязательно использует stdscr. Например, функция **printw** выводит строку данных в окно stdscr. Функция **wprintw** выводит строку в окно, заданное в аргументе *window*. Функция **mvprintw** преобразует указанные координаты в координаты stdscr, а затем выполняет ту же операцию, что и **printw** . Аналогично, функция **mvwprintw** преобразует указанные координаты в координаты указанного окна, а затем выполняет ту же операцию, что и **wprintw**.

Структура программ curses

Программы curses, как правило, выполняются в следующем порядке:

1. Запуск curses.
2. Проверка поддержки цвета (не обязательно).
3. Запуск поддержки цвета (не обязательно).
4. Создание одного или нескольких окон.
5. Управление окнами.
6. Уничтожение одного или нескольких окон.
7. Завершение curses.

Некоторые шаги не обязательны; ход работы конкретной программы не обязательно должен соответствовать этой последовательности.

Коды возврата

Большинство функций curses возвращают либо целое значение ERR, либо целое значение OK. Если функция не соответствует этому соглашению, то это всегда указывается в ее описании. Функции, возвращающие указатели, в случае ошибки возвращают пустой указатель.

Инициализация curses

В этом разделе рассмотрены команды, применяемые для инициализации curses.

Для инициализации curses применяются следующие команды:

Команда	Описание
endwin	Завершает работу библиотеки curses и очищает ее структуры данных
initscr	Инициализирует библиотеку curses и ее структуры данных
isendwin	Возвращает значение TRUE после вызова функции **endwin**, но до вызова функции **wrefresh**
newterm	Настраивает новый терминал
setupterm	Настраивает структуру TERMINAL библиотеки curses

В начале любой программы, применяющей функции curses, должен быть включен файл **curses.h**. Для этого укажите:

```
#include <curses.h>
```

Для работы с функциями управления окнами и экранами нужно предварительно вызвать функцию **initscr** или **newterm**. Эти функции сохраняют настройки терминала, а затем вызывают функцию **setupterm**, настраивающую терминал curses.

Приостановить работу библиотеки curses можно, например, с помощью системного вызова. Для восстановления работы необходимо вызвать функцию **wrefresh** или **doupdate**. Перед завершением программа, работающая с curses, должна вызвать функцию **endwin**. Эта функция восстанавливает исходные режимы терминала, перемещает курсор в левый нижний угол экрана и переводит терминал в текстовый режим.

Для большинства интерактивных полноэкранных программ требуется посимвольный метод ввода без эхоповтора. Для перехода в этот режим после вызова **initscr** вызовите функции **cbreak** и **noecho**. После этого для получения ввода можно применять следующие функции:

- Функция **nonl**.
- Функция **intrflush** со значением **stdscr** в параметре*Window* и **FALSE** в параметре *Flag*. Параметр *Window* обязателен, но его значение игнорируется. Поэтому в параметре *Window* можно указать уже существующую структуру **stdscr**.
- Функция **keypad** со значением **stdscr** в параметре *Window* и значением **TRUE** в параметре *Flag*.

Функция **isendwin** позволяет повысить эффективность работы программы путем исключения лишних вызовов **wrefresh**. Функция **isendwin** позволяет выяснить, запускалась ли функция **wrefresh** после последнего вызова **endwin**.

Окна в среде curses

Программы, использующие curses, могут управлять окнами, показанными на дисплее терминала. Размеры окон могут быть различными: окно может занимать весь экран, а может вмещать только один символ.

Примечание: Панель - это окно, которое не ограничено размерами экрана.

В программах, использующих curses, окна рассматриваются как переменные с типом WINDOW. Тип данных WINDOW определен в файле **/usr/include/curses.h** как структура данных C. При создании окна система выделяет область машинной памяти для этой структуры данных. В этой структуре хранятся все параметры окна. Изменив данные окна во внутренней памяти, программа должна запустить функцию **wrefresh** (или аналогичную ей) для обновления физического экрана. Только после этого внутренние изменения отразятся на внешнем дисплее.

Структура стандартного окна

В библиотеке curses предусмотрена структура стандартного виртуального окна, которая называется *stdscr*. В структуре stdscr хранится все содержимое дисплея. Структура окна stdscr, описывающая дисплей, создается автоматически при инициализации библиотеки curses. При этом значения *высоты* и *ширины* устанавливаются равными размерам физического дисплея.

Программы, применяющие stdscr, сначала изменяют внутреннюю структуру. Затем они могут обновить содержимое дисплея с помощью функции **refresh**.

В дополнение к stdscr, вы можете определить и другие окна. Такие окна называются *пользовательскими*. Как и stdscr, пользовательские окна представляют собой структуры, хранящиеся в памяти. Вы можете создать любое количество окон, их число ограничено лишь объемом доступной памяти. Программы, использующие curses, могут одновременно управлять стандартным окном и пользовательскими окнами.

Структура текущего окна

Помимо окна stdscr, функции curses поддерживают еще одно виртуальное окно - *curscr* (текущий экран). Окно curscr служит внутренним представлением изображения, показанного в настоящий момент на дисплее.

Если программе необходимо синхронизовать внутреннее и внешнее представление, она вызывает функцию обновления физического дисплея **wrefresh** (при работе с stdscr - **refresh**).

Окно curscr зарезервировано в curses для внутреннего применения. Изменять curscr нельзя.

Вложенные окна

Функции curses позволяют создавать *вложенные окна*. Вложенные окна - это прямоугольные фрагменты, расположенные внутри других окон. Вложенное окно тоже относится к типу данных WINDOW. Окно, содержащее вложенное окно, называется "родительским", а вложенное окно - "дочерним".

Изменения, вносимые в зоне перекрытия окон, затрагивают как родительское, так и дочернее окно. После изменения вложенного окна необходимо вызвать функцию **touchline** или **touchwin** для родительского окна перед его обновлением.

Функция	Описание
touchline	Выполняет принудительное обновление группы строк при следующем вызове функции **wrefresh** .
touchwin	Выполняет принудительное обновление всех символов в окне при следующем вызове функции **wrefresh**. Функция **touchwin** не сохраняет информацию об оптимизации. Эта функция полезна при работе с перекрывающимися окнами.

При обновлении родительского окна автоматически обновляются и все дочерние окна. Вложенное окно также может содержать дочерние окна. Процесс организации многоуровневых окон называется *вложением*.

Удалить родительское окно можно только после того, как будут удалены все дочерние окна (это можно сделать с помощью функции **delwin**). При попытке удалить окно, у которого есть дочерние окна, функция curses вернет сообщение об ошибке.

Панели

Панель - это окно, размер которого не ограничен размерами дисплея, а сама панель не связана ни с одной определенной областью дисплея. Как правило, панели бывают больше физического дисплея, поэтому в любой момент времени пользователь видит лишь часть панели.

Панели удобно использовать для представления больших массивов связанных данных, которые желательно хранить в одном окне, но не обязательно показывать все сразу.

WindowsОкна в панелях называются *вложенными панелями*. Вложенные панели находятся внутри родительских. В отличие от вложенных окон, их расположение определяется не экранными координатами, а относительными координатами родительской панели.

В отличие от других окон, содержимое панелей не обновляется автоматически при прокрутке или ввода данных с эхо-отображением на экране. Как и в случае с окнами, после изменения вложенной панели нужно сначала запустить функцию **touchline** или **touchwin** для родительской панели и только после этого обновлять ее.

К панелям применимы все функции curses, за исключением **newwin**, **subwin**, **wrefresh** и **wnoutrefresh**, вместо которых должны применяться функции **newpad**, **subpad**, **prefresh** и **pnoutrefresh**.

Управление данными в окне с помощью функций curses

При инициализации curses на экране автоматически появляется `stdscr`. Вы можете работать с этим окном, используя функции из библиотеки curses, или создать собственные окна.

Создание окон

С помощью функции **newwin** вы можете создать собственное окно.

При каждом вызове функции **newwin** библиотека curses выделяет память для размещения новой структуры, описывающей окно. Эта структура содержит всю информацию, связанную с новым окном. В библиотеке Curses число создаваемых окон не ограничено. Число вложенных окон ограничено объемом доступной памяти и задается в параметре SHRT_MAX в файле **/usr/include/limits.h**.

Вы можете изменять параметры окон вне зависимости от порядка их создания. Обновление дисплея терминала выполняется с помощью функции **wrefresh**.

Вложенные окна

Для вложенного окна необходимо задать координаты относительно дисплея. Координаты вложенного окна, созданного с помощью функции **subwin**, должны находиться в пределах родительского окна. В противном случае будет возвращено пустое значение.

Панели

Для создания панелей предусмотрены следующие функции:

Функция	Описание
newpad	Создает структуру данных, описывающую панель.
subpad	Создает и возвращает указатель на вложенную панель.

Удаление окон, панелей и вложенных окон

Координаты новой вложенной панели задаются относительно родительской панели.

Для удаления окна, панели или вложенного окна вызовите функцию **delwin**. Для удаления окна или панели необходимо предварительно удалить все дочерние элементы, в противном случае функция **delwin** вернет сообщение об ошибке.

Изменение содержимого окна или экрана

При изменении содержимого окна с помощью функций curses обновляется только внутреннее представление окна. Для изменения изображения необходимо вызвать функцию **wrefresh**. Функция **wrefresh** обновляет изображение на экране, руководствуясь информацией из структуры с описанием окна.

Обновление содержимого окна

После записи вывода в структуру с описанием окна или панели необходимо обновлять дисплей, чтобы показанное на нем изображение соответствовало внутреннему представлению. При обновлении выполняются следующие операции:

- Содержимое curscr сравнивается с содержимым пользовательского окна или stdscr
- В соответствии с полученной информацией о пользовательском окне или stdscr обновляется структура curscr
- Перерисовывается изменившаяся часть физического дисплея

Для обновления содержимого окна предназначены следующие функции:

Функция	Описание
refresh или **wrefresh**	Обновляет дисплей и curscr для отображения внесенных изменений.
wnoutrefresh и **doupdate**	Обновляет указанные окна и выводит их на дисплей. Эти функции применяются, когда требуется отобразить изменения в нескольких окнах.

Функции **refresh** и **wrefresh** сначала вызывают функцию **wnoutrefresh** для копирования обновляемого окна на текущий экран. После этого вызывается функция **doupdate** для обновления дисплея.

Кроме того, вы можете обновить одновременно несколько окон, воспользовавшись одним из следующих способов. Можно выполнить серию вызовов функции **wrefresh**, в результате чего будут попеременно вызываться функции **wnoutrefresh** и **doupdate**. Можно также вызвать функцию **wnoutrefresh** для каждого окна, а потом один раз вызвать **doupdate**. Во втором способе вывод на экран передается только один раз.

Функции перерисовки панелей

Функции **prefresh** и **pnoutrefresh** работают аналогично функциям **wrefresh** и **wnoutrefresh**.

Функция **prefresh** обновляет текущее окно и физический дисплей, а функция **pnoutrefresh** обновляет curscr для отображения изменений в пользовательской панели. Так как в данном обрабатываются не окна, а панели, необходимо указать дополнительные параметры, задающие измененные части панели и окна.

Обновление неизменных областей

Во время обновления на дисплее перерисовываются только измененные области. Остальные области можно обновить с помощью функций **touchwin** и **touchline**:

Функция	Описание
touchline	Выполняет принудительное обновление группы строк при следующем вызове функции **wrefresh** .
touchwin	Выполняет принудительное обновление всех символов в окне при следующем вызове функции **wrefresh**. Функция **touchwin** не сохраняет информацию об оптимизации. Эта функция полезна при работе с перекрывающимися окнами.

При работе с вложенными и пересекающимися окнами применяется сочетание функций **touchwin** и **wrefresh**. Для того чтобы разместить окно поверх остальных, вызовите функцию **touchwin**, а затем - функцию **wrefresh**.

Искажение окон

Если текст передается на дисплей терминала функцией, не входящей в библиотеку curses, например, **echo** или **printf**, то данные в окне могут быть показаны неправильно. В этом случае содержимое дисплея изменится, но в текущем окне эти изменения отражены не будут. При обновлении такого окна могут возникнуть ошибки. В результате этого фрагменты экрана, в которых находился неправильно показанный текст, обновлены не будут.

Аналогичная ошибка может возникнуть при перемещении окна. Изменения расположения символов, переданных на дисплей функциями не из библиотеки curses, не отражаются во внутренней структуре.

Если данные в окне показаны неправильно, вызовите функцию **wrefresh** для структуры curscr. Она восстановит соответствие между физическим дисплеем и внутренним представлением данных.

Управление содержимым окна

Для управления созданными окнами предусмотрены следующие функции:

Функция	Описание
box	Рисует рамку внутри или вокруг окна
copywin	Обеспечивает более точное управление функциями **overlay** и **overwrite**
garbagedlines	Указывает curses, что строка экрана удалена, и ее необходимо удалить с экрана перед записью информации в эту строку
mvwin	Перемещает обычное или вложенное окно
overlay и **overwrite**	Копирует одно окно поверх другого
ripoffline	Удаляет строку с экрана по умолчанию

Функции **overlay** и **overwrite** применяются в случае перекрытия окон. Функция **overwrite** уничтожает данные в отличие от функции **overlay**. Это значит, что при копировании текста из одного окна в другое с помощью функции **overwrite** пустые фрагменты целевого окна заменяются соответствующими фрагментами копируемого окна. Функция **overlay** не копирует пустые фрагменты.

Так же, как функции **overlay** и **overwrite**, функция **copywin** позволяет скопировать часть окна в другое окно. Отличие заключается в том, что для вызова функции **copywin** не требуется, чтобы окна перекрывались.

Для удаления строки из stdscr можно воспользоваться функцией **ripoffline**. Если вы передадите этой функции положительное значение в параметре *строка*, то указанное число строк в начале stdscr будет удалено. Если в параметре *строка* будет передано отрицательное значение, то будут удалены строки в нижней части окна stdscr.

Для удаления строк в заданном диапазоне перед записью в окно новой информации можно вызвать функцию **garbagedlines**.

Поддержка фильтров

Для работы с приложениями-фильтрами curses используется функция **filter**. Эта функция преобразует окно stdscr таким образом, что функции curses могут работать с ним, как с одной строкой. После вызова **filter** функции curses не применяют средства работы с терминалом, в которых требуется указывать, обработка какой строки выполняется в данный момент.

Управление курсором с помощью curses

В этом разделе рассмотрены различные типы курсоров из библиотеки curses.

В библиотеке curses есть следующие типы курсоров:

логический курсор
Определяет расположение курсора в каждом окне. Структура данных каждого окна отслеживает расположение своего логического курсора. Для каждого окна существует логический курсор.

физический курсор
Курсор дисплея. Этот курсор служит для вывода данных на дисплей рабочей станции. У каждого дисплея существует только один физический курсор.

Логический курсор в окне позволяет только выводить символы или удалять их. Для управления курсором предусмотрены следующие функции:

getbegyx
Помещает начальные координаты окна в целочисленные переменные *y* и *x*

getmaxyx
Помещает сведения о размере окна в целочисленные переменные *y* и *x*

getsyx
Выдает текущие координаты виртуального курсора дисплея

getyx
Возвращает координаты логического курсора, связанного с определенным окном

leaveok
Позволяет управлять положением физического курсора после вызова функции **wrefresh**.

перемещение
Перемещает логический курсор, связанный с stdscr.

mvcur
Перемещает физический курсор.

setsyx
Помещает виртуальный курсор дисплея в точку с заданными координатами.

wmove
Перемещает логический курсор, связанный с пользовательским окном.

После вызова функции **refresh** или **wrefresh** curses перемещает физический курсор на последний измененный символ в окне. Для того чтобы после обновления физический курсор оставался на первоначальном месте, следует выполнить функцию **leaveok**, указав нужное окно в параметре *Window* и значение **TRUE** в параметре *Flag*.

Работа с символами с помощью curses

Вводить символы в окно curses можно с клавиатуры или из приложения. В этом разделе описаны способы добавления, удаления и изменения символов в окне curses.

Размер символов

Некоторые наборы включают символы, занимающие несколько столбцов на экране.

Вывод символа, ширина которого превышает ширину целевого окна, приводит к ошибке.

Добавление символов к изображению на экране

Библиотека curses содержит набор функций, позволяющих изменять текст в окне и выделять область, которая должна быть обновлена при следующем вызове функции **wrefresh**.

Функции waddch

Функции **waddch** заменяют символ в текущей позиции логического курсора на указанный символ. После этого логический курсор перемещается на одну позицию вправо. На правой границе окна функции **waddch** автоматически добавляют символ новой строки. Кроме того, если курсор находится на нижней границе и применяется функция **scrollok**, то область прокручивается на одну строку вверх. Это происходит, например, при добавлении новой строки к последней строке окна.

При добавлении символа табуляции, новой строки или забоя библиотека curses соответственно обновляет позицию курсора в окне. Позиции табуляции расположены через каждые восемь символов. При выводе символа новой строки curses сначала вызывает функцию **wclrtoeol**, удаляющую все символы строки начиная с текущего положения курсора и до конца строки, а затем перемещает курсор. Семейство **waddch** состоит из следующих элементов:

Функция	Описание
Макрокоманда **addch**	Выводит символ в stdscr
Макрокоманда **mvaddch**	Перемещает курсор, а затем выводит символ в позиции курсора в stdscr
Макрокоманда **mvwaddch**	Перемещает курсор, а затем выводит символ в позиции курсора в пользовательское окно
Функция **waddch**	Выводит символ в пользовательское окно

Применяя совместно функции семейств **winch** и **waddch**, можно скопировать текст и атрибуты видеоизображения из одной области окна в другую. Функции семейства **winch** позволяют получать символы и атрибуты из окна или экрана, а функции **waddch** - выводить символы на экран.

Кроме того, функции **waddch** позволяют выводить управляющие символы. Эти символы будут показаны в виде ^X.

Примечание: Если позиция, для которой вызвана функция **winch**, содержит управляющий символ, тот этот символ не возвращается. Вместо него будет возвращен символ, представляющий управляющий символ на экране.

Вывод символов, отличных от управляющих

Для вывода отдельных символов, отличных от управляющих, рекомендуется использовать функции **wechochar**, которые дают значительный выигрыш в производительности. Эти функции по своему действию равносильны соответствующим парам функций **waddchr** и **wrefresh**. В семейство **wechochar** входят функция **wechochar**, макроопределение **echochar** и функция **pechochar**.

Некоторые наборы символов содержат невидимые символы. (Невидимыми называются отличные от ('\0') символы, для которых функция **wcwidth** возвращает нулевую ширину.) Приложение может выводить такие

символы в окно. Каждый невидимый символ связан с обычным символом и изменяет его определенным образом. Невидимые символы в окне не адресуются непосредственно. Они неявно адресуются при выполнении операций curses над обычными символами, с которыми связаны невидимые символы.

С невидимыми символами не связаны атрибуты. Если в интерфейсе поддерживаются широкие символы и атрибуты, то атрибуты, связанные с невидимыми широкими символами игнорируются. При выводе широких символов для всех столбцов определен единый набор атрибутов. Приложение может управлять соотношением между невидимыми символами и обычными символами с помощью интерфейса широких символов. Это соотношение зависит от конкретного набора символов.

Обычно невидимый символ действует на обычный символ (назовем его *c*) одним из следующих способов:

- Невидимый символ может изменить вид символа *c*. (Например, некоторые невидимые символы добавляют к обычным символам диакритические знаки. В то же время, некоторые символы содержат собственные диакритические знаки.)
- Невидимый символ может объединить символ *c* со следующим за ним символом. Примером может служить образование лигатур и преобразование символов в составные изображения, слова или идеограммы.

Максимальное число невидимых символов, которые могут быть связаны с одним обычным символом, зависит от конкретной реализации, но всегда не меньше 5.

Составные символы

Составной символ - это набор связанных символов, который может включать один обычный символ, а также некоторое количество невидимых символов. Видимый составной символ обязательно содержит один обычный символ. Составные символы встречаются, например, в наборе символов ISO/IEC 10646-1:1993.

Составной символ может быть выведен на экран. Если составной символ не включает обычный символ, то все его невидимые символы связываются с обычным символом, находящимся в указанной точке экрана. При считывании информации с экрана приложение получает только видимые составные символы.

Тип данных cchar_t содержит описание и представление составного символа. Если cchar_t соответствует невидимому составному символу (в котором отсутствует обычный символ), то представление символа игнорируется. При выводе используется представление видимого символа, уже показанного на экране.

Объект типа cchar_t можно инициализировать с помощью функции **setchar**, а его содержимое можно получить с помощью функции **getchar**. Результат применения функций к объекту cchar_t, который не был инициализирован указанным способом или получен от функции curses типа cchar_t, не определен.

Специальные символы

Некоторые функции могут обрабатывать специальные символы. В функциях, не перемещающих курсор на основании содержимого окна, специальные символы будут влиять только на расположение последующих символов в строке. Любая заданная впоследствии операция перемещения курсора не повлияет на положение видимого курсора. В функциях, не перемещающих курсор, эти специальные символы могут применяться для управления положением последующих символов и перемещения физического курсора.

Символ	Описание
Забой	Символ Забой перемещает курсор на один столбец к началу строки, если только он уже не находится в нулевом столбце; все дальнейшие символы добавляются с новой позиции курсора.
Возврат каретки	Символ Возврат каретки перемещает курсор в начало текущей строки. Все дальнейшие символы добавляются с новой позиции курсора.
Новая строка	В операции добавления функция curses заполняет пробелами текущую строку до конца. Затем выполняется прокрутка, курсор перемещается в начало новой строки и все последующие символы добавляются с новой позиции курсора. В операции вставки функция curses очищает текущую строку до конца с помощью функции **wclrtoeol** , а затем помещает курсор в начало следующей строки. Переход курсора на следующую строку может вызвать прокрутку, если она разрешена. Все последующие символы вставляются с новой позиции курсора. Такой способ обработки можно подавить с помощью функции **filter**.
Табуляция	Символ табуляции в тексте сдвигает следующие за ним символы до очередной позиции табуляции. По умолчанию позиции табуляции расположены в столбцах 0, 8, 16 ... В операции вставки или добавления функция curses соответственно вставляет или добавляет пробелы в последовательные столбцы до очередной позиции табуляции. Если до конца текущей строки позиции табуляции отсутствуют, выполняется перевод строки и прокрутка.

Управляющие символы

Функции curses, обрабатывающие специальные символы, заменяют их на символ '^' и обычный символ (если это буква, то она будет прописной). Функциям получения текста из окна передается указанная пара символов, а не код специального символа.

Псевдографика:

Ниже описаны переменные и функции **waddch**, с помощью которых вы можете выводить на экран символы псевдографики. Для применения символов из дополнительной части кодового набора нужно установить разряд **A_ALTCHARSET**. В противном случае будут выведены символы из основной части набора (см. столбец Символ по умолчанию в следующей таблице).

Имя переменной	Символ по умолчанию	Описание символа
ACS_ULCORNER	+	левый верхний угол
ACS_LLCORNER	+	левый нижний угол
ACS_URCORNER	+	правый верхний угол
ACS_LRCORNER	+	правый нижний угол
ACS_RTEE	+	правый тройник
ACS_LTEE	+	левый тройник
ACS_BTEE	+	нижний тройник
ACS_TTEE	+	верхний тройник
ACS_HLINE	-	горизонтальная линия
ACS_VLINE	|	вертикальная линия
ACS_PLUS	+	знак плюс
ACS_S1	-	верхняя линия
ACS_S9	_	нижняя линия
ACS_DIAMOND	+	ромб
ACS_CKBOARD	:	штриховка
ACS_DEGREE	,	символ градуса
ACS_PLMINUS	#	знак плюс-минус
ACS_BULLET	o	жирная точка
ACS_LARROW	<	стрелка влево

Имя переменной	Символ по умолчанию	Описание символа
ACS_RARROW	>	стрелка вправо
ACS_DARROW	v	стрелка вниз
ACS_UARROW	^	стрелка вверх
ACS_BOARD	#	жирная штриховка
ACS_LANTERN	#	"солнышко"
ACS_BLOCK	#	закрашенный квадрат

Функции waddstr

Функции **waddstr** добавляют в окно строку, оканчивающуюся символом NULL, начиная с текущего символа. Для добавления одного символа эффективнее функция **waddch**. Функции **waddstr** предназначены для вывода строк символов. Семейство **waddstr** состоит из следующих элементов:

Функция	Описание
Макрокоманда **addstr**	Выводит строку символов в stdscr
Макрокоманда **mvaddstr**	Перемещает логический курсор в заданную позицию, а затем добавляет строку символов в stdscr
Функция **waddstr**	Выводит строку символов в пользовательское окно
Макрокоманда **wmvaddstr**	Перемещает логический курсор в заданную позицию, а затем выводит строку символов в пользовательское окно

Функции winsch

Функции **winsch** вставляют символ перед текущим символом в окне. Все символы справа от вставленного сдвигаются на один столбец вправо. Такой сдвиг может привести к потере крайнего правого символа. Положения логических и физических курсоров не меняются. Семейство **winsch** включает следующие элементы:

Функция	Описание
Функция **insch**	Вставляет символ в stdscr
Макрокоманда **mvinsch**	Перемещает логический курсор в указанную позицию, а затем вставляет символ в stdscr
Макрокоманда **mvwinsch**	Перемещает логический курсор в указанную позицию, а затем вставляет символ в пользовательское окно
Функция **winsch**	Вставляет символ в пользовательское окно

Функции winsertln

Функции **winsertln** вставляют пустую строку перед следующей. Функция **insertln** вставляет строку в stdscr. Нижняя строка окна удаляется. Функция **winsertln** выполняет аналогичную операцию с пользовательским окном.

Функции wprintw

Функции **wprintw** заменяют последовательность символов (начиная с текущего) форматированным выводом. Применяется тот же формат, что и в команде **printf**. Семейство **printw** состоит из следующих элементов:

Функция	Описание
Макрокоманда **mvprintw**	Перемещает логический курсор в указанную позицию, а затем заменяет последовательность символов в stdscr
Макрокоманда **mvwprintw**	Перемещает логический курсор в указанную позицию, а затем заменяет последовательность символов в пользовательском окне
Макрокоманда **printw**	Заменяет последовательность символов в stdscr
Функция **wprintw**	Заменяет последовательность символов в пользовательском окне

Функции **wprintw** для замены символов вызывают функции **waddch**.

Макрокоманда unctrl

Макрокоманда **unctrl** возвращает печатное представление управляющих символов в формате ^X. Печатные символы макрокоманда **unctrl** оставляет без изменений.

Прокрутка текста

Следующие функции предназначены для прокрутки:

Функция	Описание
idlok	Разрешает библиотеке curses применять аппаратные функции вставки и удаления строк
scrollok	Разрешает прокрутку окна при выходе курсора за правую границу последней строки
setscrreg и **wsetscrreg**	Задает область программной прокрутки в окне

Если пользователь или приложение перемещают курсор за нижний край окна, то может быть выполнена прокрутка. Прокрутка должна быть разрешена функцией **scrollok**. Прокрутка выполняется, если она разрешена указанной функцией и происходит одно из следующих событий:

- Курсор выходит за край окна.
- К последней строке добавляется символ новой строки.
- В последнюю позицию последней строки вставляется символ.

При прокрутке окна функции curses обновляют как окно, так и экран. Однако для того чтобы прокрутка была видна, необходимо вызвать функцию **idlok** с параметром *Флаг*, равным **TRUE** (истина).

Если прокрутка запрещена, то курсор остается на последней строке в позиции последнего введенного символа.

Если прокрутка окна разрешена, то с помощью функций **setscrreg** в нем можно создать программную область прокрутки. Вызываемым функциям **setscrreg** необходимо передать номера верхней и нижней строк области. После этого любая попытка вывести курсор за указанную нижнюю строку будет приводить к прокрутке всей области на одну строку вверх. Макроопределение **setscrreg** позволяет создать область прокрутки в stdscr. Для создания областей прокрутки в пользовательских окнах применяется функция **wsetscrreg**.

Примечание: В отличие от функции **idlok**, функции **setscrreg** не используют аппаратную прокрутку, даже если терминал обладает такой возможностью.

Удаление символов

Удаление символов может быть выполнено путем замены их пробелами или сдвига строки влево.

Функции werase

Макрокоманда **erase** заменяет пробелами все символы stdscr. Функция **werase** заменяет пробелами все символы пользовательского окна. Для удаления одного символа из окна воспользуйтесь функцией **wdelch**.

Функции wclear

Следующие функции предназначены для очистки экрана:

Функция	Описание
clear или **wclear**	Очищает экран и устанавливает флаг clear для следующего обновления.
clearok	Определяет, будет ли выполнена очистка окна при следующем вызове функции **refresh** или **wrefresh**.

Функции **wclear** аналогичны функциям **werase**. Однако после замены всех символов окна пробелами функции **wclear** дополнительно вызывают функцию **wclearok**. В результате при следующем вызове функции **wrefresh** экран автоматически очищается.

Семейство функций **wclear** включает функцию **wclear**, макрокоманду **clear** и функцию **clearok**. Макрокоманда **clear** заменяет пробелами все символы stdscr.

Функции wclrtoeol

Макрокоманда **clrtoeol** работает с stdscr, тогда как функция **wclrtoeol** выполняет те же действия в пользовательском окне.

Функции wclrtobot

Макрокоманда **clrtobot** работает с stdscr, тогда как функция **wclrtobot** выполняет те же действия в пользовательском окне.

Функции wdelch

Следующие функции предназначены для удаления символов с экрана:

Функция	Описание
Макрокоманда **delch**	Удаляет текущий символ из stdscr
Макрокоманда **mvdelch**	Перемещает логический курсор в указанную позицию, а затем удаляет символ из stdscr
Макрокоманда **mvwdelch**	Перемещает логический курсор в указанную позицию, а затем удаляет символ из пользовательского окна
Функция **wdelch**	Удаляет текущий символ из пользовательского окна

Функции **wdelch** удаляют текущий символ и смещают все последующие символы текущей строки на одну позицию влево. Последний символ строки заменяется пробелом. Семейство функций **delch** состоит из следующих элементов:

Функции wdeleteln

Функции **deleteln** удаляют текущую строку и перемещают все последующие строки на одну строку вверх. Нижняя строка окна при этом очищается.

Получение символов

Программы могут получать символы как с клавиатуры, так и с экрана. Функции **wgetch** получают символы с клавиатуры, а функции **winch** - с экрана.

Функции wgetch

Функции **wgetch** получают символы, введенные с клавиатуры, подключенной к текущему терминалу. Если содержимое окна или положение курсора изменилось, то перед получением символа эти функции вызывают функции **wrefresh**. Дополнительная информация приведена в описании функции **wgetch** в *Technical Reference: Base Operating System and Extensions, Volume 2*.

Семейство **wgetch** состоит из следующих элементов:

Функция	**Описание**
Макрокоманда **getch**	Получает символ из stdscr
Макрокоманда **mvgetch**	Перемещает логический курсор в указанную позицию, а затем получает символ из stdscr
Макрокоманда **mvwgetch**	Перемещает логический курсор в указанную позицию, а затем получает символ из пользовательского окна
Функция **wgetch**	Получает символ из пользовательского окна

Для того чтобы поместить символ, полученный с помощью функции **wgetch**, обратно в очередь ввода, вызовите функцию **ungetch**. Этот символ будет получен при следующем вызове функции **wgetch**.

Режимы работы терминала

Результат работы функции **wgetch** зависит от режима работы терминала. Ниже описаны действия, выполняемые функциями **wgetch** в каждом из режимов:

Функция	**Описание**
Режим **DELAY**	Приостанавливает считывание до тех пор, пока программа не обработает текст. Если включен режим CBREAK, то программа останавливается после чтения одного символа. Если режим CBREAK выключен (режим NOCBREAK), то функция **wgetch** прекращает чтение символов после обнаружения первого символа новой строки. Если включен режим ECHO, считанные символы отображаются в окне.
Режим **HALF-DELAY**	Приостанавливает считывание до ввода символа или наступления заданного тайм-аута. Если включен режим ECHO, считанные символы отображаются в окне.
Режим **NODELAY**	Возвращает значение ERR, если ввод отсутствует.

Примечание: В случае применения функций **wgetch** не устанавливайте одновременно режимы NOCBREAK и ECHO. Это может привести к непредсказуемым результатам, зависящим от состояния драйвера терминала в момент ввода каждого символа.

Функциональные клавиши

Функциональные клавиши определены в файле **curses.h**. Они могут быть получены функцией **wgetch**, если будут набраны с дополнительной клавиатуры. Терминал может поддерживать не все функциональные клавиши. Информация о поддержке клавиш приведена в базе данных **terminfo**. Список функциональных клавиш приведен в описании функций getch, mvgetch, mvwgetch и wgetch , приведенном в книге *Technical Reference: Base Operating System and Extensions, Volume 2*.

Получение кодов функциональных клавиш

Если программа вызывает функцию **keypad** и пользователь нажимает функциональную клавишу, то вместо соответствующей этой клавише последовательности символов возвращается код нажатой функциональной клавиши. Файл **/usr/include/curses.h** содержит определения возможных функциональных клавиш. Каждое определение начинается с префикса **KEY_**, а коды клавиш представляют собой целые числа, начинающиеся с 03510.

При получении символа, который может быть началом последовательности символов функциональной клавиши (например, символа Escape), curses устанавливает таймер (структуру типа timeval, определенную в файле **/usr/include/sys/time.h**). Если оставшаяся часть последовательности не поступает за отведенное время, то символ передается для дальнейшей обработки. В противном случае возвращается код функциональной клавиши. По этой причине программы реагируют на нажатие клавиши Esc с задержкой. В связи с этим не

рекомендуется назначать какие-либо действия на клавишу Esc, если ввод считывается с клавиатуры функцией **wgetch**. Параметры таймера можно изменить с помощью переменной среды **ESCDELAY**.

Переменная среды **ESCDELAY** задает время ожидания дополнительных символов, по истечении которого вызывающей программе код клавиши Escape. **ESCDELAY=1** означает задержку в 1/5000 с. Если переменная **ESCDELAY** равна нулю, то система передает код клавиши Escape, не ожидая появления в буфере дополнительной информации. Допустимы значения от 0 до 99999. По умолчанию значение **ESCDELAY** равно 500 (0,1 с).

Для того чтобы запретить функции **wgetch** устанавливать таймер, вызовите функцию **notimeout**. В этом случае curses не будет делать различия между функциональными и обычными клавишами.

Функция keyname

Функция **keyname** возвращает указатель на имя клавиши, код которой указан в аргументе *Клавиша*. Значение *Клавиша* может быть получено от любой из функций **wgetch**, **getch**, **mvgetch** или **mvwgetch**.

Функции winch

Функции **winch** считывают текущий символ. Если с символом связаны атрибуты, то они объединяются с полученным значением с помощью двоичной дизъюнкции. С помощью функций **winch** можно узнать код символа или его атрибуты. Для этого следует вычислить значение **A_CHARTEXT** & **A_ATTRIBUTES**. Указанные константы определены в файле **curses.h**. Семейство **winch** содержит следующие элементы:

Функция	Описание
Макрокоманда **inch**	Получает текущий символ из stdscr
Макрокоманда **mvinch**	Перемещает логический курсор, а затем считывает символ из stdscr с помощью функции **inch**
Макрокоманда **mvwinch**	Перемещает логический курсор, а затем вызывает функцию **winch** для считывания символа из пользовательского окна
Функция **winch**	Получает текущий символ из пользовательского окна

Функции wscanw

Функции **wscanw** считывают символьные данные, интерпретируют их в соответствии с заданной спецификацией преобразования и сохраняют результат. Для считывания символов функции **wscanw** пользуются функциями **wgetstr**. Семейство **wscanw** состоит из следующих элементов:

Функция	Описание
Макрокоманда **mvscanw**	Перемещает логический курсор, а затем получает и обрабатывает данные из stdscr
Макрокоманда **mvwscanw**	Перемещает логический курсор, а затем получает и обрабатывает данные из пользовательского окна
Макрокоманда **scanw**	Получает и обрабатывает данные из stdscr
Функция **wscanw**	Получает и обрабатывает данные из пользовательского окна

Функция **vwscanw** просматривает содержимое окна, получая список аргументов переменной длины. Информация о работе со списками аргументов переменной длины приведена в описании макрокоманд **varargs** в книге *Technical Reference: Base Operating System and Extensions, Volume 2*.

Основные сведения о функциях curses для работы с терминалом

Возможности пользовательской программы ограничены, в частности, характеристиками терминала, на котором она выполняется.

В этом разделе содержится информация об инициализации терминалов и определении их характеристик.

Работа с несколькими терминалами

С помощью функций curses можно одновременно выполнять ввод-вывод на нескольких терминалах. Эти функции позволяют создавать новые терминалы, переключать ввод и вывод, а также получать информацию о характеристиках терминалов.

Для запуска curses на одном экране по умолчанию вызовите функцию **initscr**. Если ваше приложение выводит данные на несколько терминалов, то следует воспользоваться функцией **newterm**. Функцию **newterm** нужно вызвать для каждого терминала. Функцию **newterm** следует применять и в том случае, если ваше приложение обрабатывает ошибки таким образом, что на терминалах, не поддерживающих полноэкранный режим, оно может продолжить работу в построчном режиме.

По окончании работы программа должна вызвать функцию **endwin** для каждого терминала, который она использовала. Если было открыто несколько новых терминалов (функция **newterm** была вызвана несколько раз), то эти терминалы должны закрываться (посредством вызовов функции **endwin**) в обратном порядке.

Функция **set_term** позволяет назначить терминал для операций ввода и вывода.

Определение характеристик терминала

Для определения характеристик терминала предусмотрены следующие функции curses:

Функция	Описание
has_ic	Определяет, поддерживает ли терминал вставку символов
has_il	Определяет, поддерживает ли терминал вставку строк
longname	Возвращает полное имя терминала

Функция **longname** возвращает указатель на статическую область памяти, в которой хранится подробное описание текущего терминала. Определение статической области создается только при вызове функции **initscr** или **newterm**. Если вы хотите использовать функцию **longname** при работе с несколькими терминалами, то учтите, что при каждом вызове функции **newterm** содержимое этой области заменяется. Вызов функции **set_term** не восстанавливает прежнее содержимое. В связи с этим следует сохранять содержимое этой области в промежутках между вызовами функции **newterm**.

Функция **has_ic** возвращает значение TRUE, если терминал может вставлять и удалять символы.

Функция **has_il** возвращает значение TRUE, если терминал может вставлять и удалять строки, либо может имитировать эти действия с помощью областей прокрутки. Функция **has_il** позволяет определить, можно ли включить физическую прокрутку функцией **scrollok** или **idlok**.

Выбор режимов ввода и вывода для терминала

Функции управления вводом-выводом определяют, каким образом приложение считывает данные и предоставляет информацию пользователю.

Режимы ввода

К специальным символам ввода относятся символы управления потоком, символ прерывания, символ стирания и символ уничтожения. В библиотеке curses реализованы следующие взаимоисключающие режима ввода, что позволяет приложениям гибко управлять вводом символов:

Обычный режим

Обычная обработка текста по строкам, все специальные символы обрабатываются вне приложения. Этот режим аналогичен стандартному режиму ввода. При переходе в этот режим путем вызова функции nocbreak( ) флаги ISIG и IXON не изменяются. Однако они устанавливаются при переходе в этот режим путем вызова функции noraw( ).

В данной реализации допускается применение символов стирания и уничтожения во всех поддерживаемых локалях, независимо от ширины символов.

cbreak, режим

Введенные пользователем символы немедленно поступают в приложение, функции curses не выполняют специальную обработку символов стирания и уничтожения. В приложении можно установить режим cbreak для того, чтобы оно само редактировало строки, но при этом его выполнение можно будет прервать с помощью клавиш отмены задания. Этот режим аналогичен нестандартному режиму ввода, случай B (MIN равен 1, ICRNL сброшен). При переходе в этот режим значения флагов ISIG и IXON не изменяются.

Режим Half-delay

Аналогичен режиму **cbreak**, за исключением того, что функции ввода ожидают либо завершения интервала, определенного в приложении, либо момента, когда символ станет доступным, в зависимости от того, что произойдет раньше. Этот режим аналогичен нестандартному режиму ввода, случай C (TIME равно значению, указанному в приложении). При переходе в этот режим значения флагов ISIG и IXON не изменяются.

Режим прямого ввода

В этом режиме приложению предоставляются максимальные возможности по управлению вводом с терминала. В приложение передаются все введенные с клавиатуры символы. Этот режим аналогичен нестандартному режиму обработки ввода, случай D. При переходе в этот режим флаги ISIG и IXON сбрасываются.

При инициализации curses с помощью функции **initscr** или **newterm** текущие параметры терминала сохраняются. В дальнейшем их можно восстановить с помощью функции **endwin**. Начальный режим ввода при работе с функциями curses не определен, если только приложение не поддерживает дополнительные возможности curses - тогда в начале работы устанавливается режим **cbreak**.

Эффект нажатия клавиши BREAK зависит от других разрядов в драйвере дисплея, на которые функции curses не влияют.

Режим задержки

Следующие взаимоисключающие режимы задержки определяют, насколько быстро происходит возврат из некоторых функций curses, если отсутствует ввод терминала, ожидающий вызова функции:

С задержкой	**Описание**
Без задержки	Происходит сбой функции.
С задержкой	Приложение ожидает передачи текста в приложение. В режиме cbreak или в режиме прямого ввода ожидание заканчивается после приема одного символа. В других режимах приложение ждет до получения первого символа новой строки.

Обработка функциональных клавиш в режиме "Без задержки" не определена.

Режим эхоповтора

Режим эхоповтора определяет, должны ли функции curses отображать вводимые символы на экране. Работа в режиме эхоповтора аналогична обработке ввода при установленном флаге echo в поле local mode структуры **termios**, соответствующей терминалу данного окна. Однако функции curses при вызове всегда сбрасывают флаг echo, чтобы операционная система не выполняла эхоповтор. Способ эхоповтора символов в curses отличен от способа эхоповтора в операционной системе, так как curses выполняют дополнительную обработку ввода, поступающего на терминал.

В режиме эхоповтора функции curses самостоятельно отображают символы. Вызванная приложением функция ввода сохраняет все введенные видимые символы в текущем или указанном окне в позиции курсора, как при вызове функции **addch**. Выполняются и все дополнительные действия по обработке ввода, например, перемещение курсора и перенос строк.

В режиме без эхоповтора отображение символов должно выполняться приложением. Приложения часто отображают ввод в своей области экрана или вообще отключают эхоповтор.

На некоторых синхронных и сетевых асинхронных терминалах эхоповтор нельзя отключить, так как он выполняется самими терминалами. При разработке приложений для таких терминалов следует учитывать, что все вводимые символы будут появляться на экране в текущей позиции курсора.

Следующие функции связаны с режимом эхоповтора:

Функция	Описание
cbreak или **nocbreak**	Переключает терминал в режим CBREAK или выключает этот режим
delay_output	Задает интервал задержки вывода в миллисекундах
echo и **noecho**	Включает или выключает эхоповтор вводимых символов
halfdelay	Возвращает ERR, если в течение указанного времени после блокировки не было введено ни одного символа
nl или **nonl**	Определяет, должны ли функции curses преобразовывать символ начала строки в символы возврата каретки и перевода строки (при выводе) и символ Return в символ начала строки (при вводе)
raw или **noraw**	Переключает терминал в режим прямого ввода или выключает этот режим

Функция **cbreak** включает режим, частично совпадающий с режимом прямого ввода (функция **raw**). В режиме cbreak введенные пользователем символы немедленно передаются в программу, символы стирания и уничтожения не обрабатываются. Однако, в отличие от режима прямого ввода, выполняется обработка символов прерывания и управления потоком. Кроме того, введенные символы хранятся в буфере терминала до ввода символа начала строки или возврата каретки.

Примечание: В режиме CBREAK драйвер терминала не выполняет преобразование символов.

Функция **delay_output** задает время задержки вывода в миллисекундах. Не следует злоупотреблять вызовами этой функции, так как она генерирует символы заполнения, а не отсчитывает паузу по тактам процессора.

Функция **echo** переводит терминал в режим эхоповтора. В этом режиме функции curses отображают введенные с клавиатуры символы в текущей позиции курсора на экране. Функция **noecho** выключает режим эхоповтора.

Функция **nl** определяет, должны ли функции curses при выводе заменять символы новой строки на символы возврата каретки и перевода строки. Функция **nonl** определяет, должны ли функции curses при вводе заменять символы возврата каретки в символы новой строки. По умолчанию выполняются обе эти операции. При отключении этих преобразований библиотека функций curses получает возможность более гибко управлять переводом строк, вследствие чего возрастает скорость движения курсора.

Функция **nocbreak** выключает режим cbreak.

Функция **raw** переключает терминал в режим прямого ввода. В этом режиме введенные пользователем символы немедленно передаются в программу. Символы прерывания, выхода, приостановки и управления потоком также передаются в программу, а не вызывают генерацию сигнала, как в режиме cbreak. Функция **noraw** отключает режим прямого ввода.

Работа с файлами terminfo и termcap

При инициализации библиотеки curses она определяет тип терминала с помощью значения переменной среды **TERM**. Затем curses выполняет поиск определения, в котором содержится информация о характеристиках терминала. Обычно эти данные хранятся в локальном каталоге, имя которого указано в

переменной среды **TERMINFO**, или в каталоге **/usr/share/lib/terminfo**. Все программы curses сначала проверяют каталог, указанный в переменной **TERMINFO**, если она определена. Если она не определена, проверяется каталог **/usr/share/lib/terminfo**.

Например, если переменной среды **TERM** присвоено значение `vt100`, а переменной **TERMINFO** - значение **/usr/mark/myterms**, то curses проверяет содержимое файла **/usr/mark/myterms/v/vt100**. Если этот файл не существует, curses проверяет содержимое файла **/usr/share/lib/terminfo/v/vt100**.

Если вам требуется переопределить значения, заданные в описании терминала, установите переменные среды **LINES** и **COLUMNS**.

Применение функций terminfo в программах

Функции **terminfo** позволяют программе напрямую обращаться к базе данных terminfo. Например, с их помощью можно программировать функциональные клавиши. Во всех остальных случаях рекомендуется использовать функции из библиотеки curses.

Инициализация терминалов

Сначала программа должна вызвать функцию **setupterm**. Как правило, эта функция вызывается неявно при вызове функции **initscr** или **newterm**. Функция **setupterm** из базы данных **terminfo** значения переменных, зависящих от терминала. База данных **terminfo** содержит булевские, численные и строковые переменные. Значения всех таких переменных **terminfo** зависят от того, какой терминал выбран. После считывания переменных из базы данных функция **setupterm** помещает описание терминала в **cur_term**. При работе с несколькими терминалами можно с помощью функции **set_curterm** определить переменную **cur_term** для данного терминала.

Функция **restartterm** во многом схожа с функцией **setupterm**. Однако она вызывается после восстановления исходного состояния памяти. Например, ее рекомендуется вызывать после функции **scr_restore**. Функция **restartterm** "полагает", что опции ввода и вывода те же, что и в момент сохранения памяти, но что тип терминала и скорость передачи данных могли измениться.

Функция **del_curterm** освобождает память, выделенную для хранения информации о характеристиках указанного терминала.

Файлы заголовков

Файлы **curses.h** и **term.h** должны быть указаны в программе в следующем порядке:

```
#include <curses.h>
#include <term.h>
```

Эти файлы содержат определения строковых и числовых переменных, а также флагов базы данных **terminfo**.

Обработка характеристик терминала

Передайте все строки с параметрами функции **tparm** для инициализации. Для того чтобы напечатать все строки **terminfo** и вывод функции **tparm**, вызовите функцию **tputs** или **putp**.

Функция	Описание
putp	Обеспечивает быстрый доступ к функции **tputs**
tparm	Инициализирует строку с параметрами
tputs	Добавляет символы заполнения к строке и выводит ее

Следующие функции можно применять для считывания характеристик терминала и передачи их в другие функции:

Функция	Описание
tigetflag	Возвращает значение указанной булевской переменной. Если указанная переменная другого типа, возвращает -1.
tigetnum	Возвращает значение указанной числовой переменной. Если указанная переменная другого типа, возвращает -2.
tigetstr	Возвращает значение указанной строковой переменной. Если указанная переменная другого типа, возвращает значение (**char** *) -1.

Завершение работы программы

При завершении работы программы необходимо восстановить исходное состояние терминала. Для этого следует вызвать функцию **reset_shell_mode**. Если программа работает с адресацией курсора, то она при запуске должна вывести строку **enter_ca_mode**, а при завершении работы - строку **exit_ca_mode**.

Если в программе выполняется временный вход в оболочку, то перед запуском оболочки необходимо вызвать функцию **reset_shell_mode** и вывести строку **exit_ca_mode**. После завершения сеанса оболочки программа должна вывести строку **enter_ca_mode** и вызвать функцию **reset_prog_mode**. Эта процедура отличается от стандартного выхода из программы с помощью функции curses **endwin**.

Низкоуровневые функции для работы с экраном

Следующие функции можно применять для выполнения низкоуровневых операций с экраном:

Функция	Описание
ripoffline	Удаляет одну строку из stdscr
scr_dump	Записывает содержимое виртуального экрана в указанный файл
scr_init	Инициализирует структуры данных curses с помощью значений из указанного файла
scr_restore	Выводит на виртуальный экран содержимое, сохраненное в файле

Функции termcap

Если программа работает с информацией о терминале из файла **termcap**, то для ее преобразования применяются функции **termcap**. Параметры функций **termcap** те же. Библиотека curses эмулирует эти функции с помощью базы данных **terminfo**. Поддерживаются следующие функции **termcap**:

Функция	Описание
tgetent	Эмулирует функцию **setupterm**.
tgetflag	Возвращает запись идентификатора termcap булевского типа.
tgetnum	Возвращает запись идентификатора termcap числового типа.
tgetstr	Возвращает запись идентификатора termcap строкового типа.
tgoto	Дублирует функцию **tparm**. Вывод функции **tgoto** должен передаваться в функцию **tputs**.

Преобразование описаний termcap в описания terminfo

Команда **captoinfo** преобразует описания **termcap** в описания **terminfo**. Приведенный ниже пример иллюстрирует работу команды **captoinfo**:

```
captoinfo
/usr/lib/libtermcap/termcap.src
```

Эта команда преобразует файл **/usr/lib/libtermcap/termcap.src** в исходный файл **terminfo**. Команда **captoinfo** записывает результаты работы в стандартный вывод, при этом комментарии и прочая подобная информация в файле сохраняются.

Работа с терминалами

Перечисленные ниже функции предназначены для сохранения и восстановления состояния режимов терминала:

Функции	Описание
savetty	Сохраняет текущее состояние режимов терминала.
resetty	Восстанавливает состояние режимов терминала, существовавшее на момент последнего вызова функции **savetty**.

Синхронные и сетевые асинхронные терминалы

Синхронные, сетевые синхронные (NWA) и нестандартные напрямую подключенные асинхронные терминалы обычно применяются при работе с мейнфреймами и обмениваются данными с хостом в блочном режиме. На таких терминалах пользователь вводит данные, а затем нажимает специальную клавишу для передачи введенных данных на хост.

Примечание: В некоторых случаях, несмотря на то, что можно пересылать данные на хост блоками произвольной длины, начинать пересылку данных по нажатию одной клавиши невозможно или нежелательно, так как это может нарушить работу приложений, основанных на посимвольном вводе.

Вывод

Интерфейс curses можно применять для выполнения любых операций вывода на терминал, но с одним возможным исключением: на некоторых терминалах применение функции **refresh** для обновления фрагмента экрана может вызвать перерисовку всего экрана.

Если перед каждой такой операцией необходимо предварительно очищать экран, то результаты могут быть далеки от желаемого.

Ввод

Вследствие особого характера работы с синхронными (блочными) и сетевыми асинхронными терминалами могут поддерживаться не все функции ввода из библиотеки curses. В частности, отметим следующее:

- На некоторых терминалах невозможен посимвольный ввод. Для передачи введенных символов на хост нужно нажимать специальную клавишу.
- На некоторых терминалах невозможно отключить эхоповтор, так как его выполняет сам терминал. При разработке приложений с применением функций curses для таких терминалов следует учитывать, что все вводимые символы будут появляться на экране в текущей позиции курсора, которая не всегда совпадает с позицией курсора в окне.

Работа с цветными символами

С помощью процедур управления цветом вы можете использовать в программах curses дополнительные возможности цветных терминалов.

Перед изменением параметров цветов следует проверить, поддерживается ли соответствующий режим. Для этого можно воспользоваться функцией **has_colors** или **can_change_color**. Функция **can_change_color** также позволяет проверить, может ли программа изменять определения цветов терминала. Аргументы для этих функций не требуются.

Функция	Описание
can_change_color	Проверяет, поддерживает ли данный терминал выбранные цвета и можно ли изменять определения цветов
has_colors	Проверяет, поддерживает ли данный терминал выбранные цвета
start_color	Инициализирует восемь основных цветов и две глобальные переменные, **COLORS** и **COLOR_PAIRS**

Если терминал поддерживает выбранные цвета, то перед вызовом каких-либо функций управления цветами следует вызвать функцию **start_color**. Рекомендуется вызывать эту функцию сразу после функции **initscr** и после успешной проверки цветов. Глобальная переменная **COLORS** задает максимальное количество цветов, поддерживаемых терминалом. Глобальная переменная **COLOR_PAIRS** задает максимальное количество поддерживаемых пар цветов.

Работа с атрибутами изображения

Вы можете задать в программе некоторые атрибуты изображения, показываемого на мониторе.

Атрибуты изображения, битовые маски и цвета по умолчанию

Функции Curses позволяют изменять следующие атрибуты:

A_ALTCHARSET
Альтернативная кодовая страница.

A_BLINK
Мигание.

A_BOLD
Выделение яркостью или жирным шрифтом.

A_DIM
Затемненное изображение.

A_NORMAL
Стандартные атрибуты.

A_REVERSE
Негативное изображение.

A_STANDOUT
Предпочтительный режим выделения информации на терминале.

A_UNDERLINE
Подчеркивание.

COLOR_PAIR (Номер)
Показывает пару цветов, соответствующую указанному *Номеру*. Эта пара цветов должна быть предварительно инициализирована с помощью функции **init_pair**.

Эти атрибуты определены в файле **curses.h**. Атрибуты можно либо передавать в функции **wattron**, **wattroff** и **wattrset**, либо объединять их с помощью дизъюнкции с символами, передаваемыми функции **waddch**. Поразрядная дизъюнкция в языке C обозначается символом | (вертикальная черта). Можно также использовать следующие битовые маски:

A_ATTRIBUTES
Извлекает атрибуты

A_CHARTEXT
Извлекает символ

A_COLOR
Извлекает информацию из поля, задающего пару цветов

A_NORMAL
Сбрасывает все атрибуты изображения

Для работы с парами цветов предусмотрены следующие макрокоманды: **COLOR_PAIR(** *номер***)** и **PAIR_NUMBER(***атрибут***)**. Макрокоманда **COLOR_PAIR**(*Номер***)** и маска **A_COLOR** применяются макрокомандой **PAIR_NUMBER(** *Атрибут***)** для получения номера пары цветов из атрибутов, задаваемых параметром *Атрибут*.

В программы, работающие с цветами, нужно включать файл **curses.h**. Он содержит следующие определения цветов, применяемых по умолчанию:

Цвет	Целое число
COLOR_BLACK (черный)	0
COLOR_BLUE (синий)	1
COLOR_GREEN (зеленый)	2
COLOR_CYAN (бирюзовый)	3
COLOR_RED (красный)	4
COLOR_MAGENTA (малиновый)	5
COLOR_YELLOW (желтый)	6
COLOR_WHITE (белый)	7

В функциях curses предполагается, что для всех терминалов фоновый цвет по умолчанию равен 0 (**COLOR_BLACK** - черный).

Установка атрибутов изображения

Текущие атрибуты окна задают вид символов, выводимых с помощью функций **addch**. Эти атрибуты являются параметрами символов. Они хранятся во время выполнения операций терминала.

attroff или wattroff
Сбрасывает атрибуты

attron или wattron
Устанавливает атрибуты

attrset или wattrset
Задает текущие атрибуты окна

standout, wstandout, standend или wstandend
Включает или выключает предпочтительный режим выделения информации на экране

vidputs или vidattr
Показывает строку, переводящую терминал в режим применения атрибутов изображения

Функция **attrset** задает текущие атрибуты экрана по умолчанию. Функция **wattrset** задает текущие атрибуты пользовательского окна.

С помощью функций **attron** и **attroff** можно устанавливать и сбрасывать отдельные атрибуты stdscr. Функции **wattron** и **wattroff** выполняют те же действия для пользовательских окон.

Функция **standout** равносильна функции **attron** с атрибутом **A_STANDOUT**. Она активирует для окна stdscr режим предпочтительного выделения изображения терминала. Функция **wstandout** равносильна функции **wattron(***Окно***, A_STANDOUT)**. Она активирует для пользовательского окна режим предпочтительного выделения изображения терминала. Функция **standend** равносильна функции **attrset(0)**. Она сбрасывает все атрибуты для stdscr. Функция **wstandend** равносильна функции **wattrset(***Окно* **, 0)**. Она сбрасывает все атрибуты для указанного окна.

Функция **vidputs** выдает строку изменения атрибутов терминала. Символы этой строки выводятся с помощью функции **putc**. Функция **vidattr** эквивалентна функции **vidputs** за исключением того, что символы выводятся с помощью функции **putchar**.

Работа с параметрами цветов

В файле **curses.h** определена макрокоманда **COLOR_PAIR** (*Номер*), поэтому вы можете управлять атрибутами цветов так же, как и прочими атрибутами. Перед работой с парой цветов ее нужно инициализировать с помощью функции **init_pair**. Для функции **init_pair** предусмотрены следующие параметры: *Пара*, *Текст* и *Фон*. Для параметра *Пара* допустимы значения от 1 до **COLOR_PAIRS**-1. Для параметров *Текст* и *Фон* допустимы значения от 0 до **COLORS**-1. Например, инициализация пары цветов 1 с черным текстом на бирюзовом фоне выполняется следующим образом:

```
init_pair(1, COLOR_BLACK, COLOR_CYAN);
```

После этого можно задать следующие атрибуты окна:

```
wattrset(win, COLOR_PAIR(1));
```

Если вы теперь выведете на экран строку Добавить цвет, то она будет показана черным цветом на бирюзовом фоне.

Получение атрибутов

С помощью значения, выданного функцией **winch**, вы можете получить информацию об атрибутах, в том числе номера пар цветов. В следующем примере значение, возвращаемое вызовом функции **winch** с логическим оператором AND (в языке C он обозначается знаком &) и битовой маской **A_ATTRIBUTES**, применяется для выделения атрибутов текущей позиции окна. Результат этой операции используется макроопределением **PAIR_NUMBER** для определения номера цветов, после чего на экране печатается этот номер - 1.

```
win = newwin(10, 10, 0, 0);
init_pair(1, COLOR_RED, COLOR_YELLOW);
wattrset(win, COLOR_PAIR(1));
waddstr(win, "apple");

number = PAIR_NUMBER((mvwinch(win, 0, 0) & A_ATTRIBUTES));
wprintw(win, "%d\n", number);
wrefresh(win);
```

Визуальные и звуковые сигналы

В библиотеку curses включены следующие функции отправки сигналов пользователям:

beep
Издает на терминале звуковой сигнал.

flash
Визуальный сигнал терминала.

Настройка curses

Все опции curses изначально сброшены, поэтому сбрасывать их перед вызовом функции **endwin** не требуется. Следующие функции позволяют задавать различные опции curses:

curs_set
Задает степень прорисовки курсора: невидимый, обычный, выделенный.

idlok
Указывает, будет ли применяться функциями curses аппаратная вставка и удаление строк на тех терминалах, где она поддерживается.

intrflush
Задает, будет ли сброшен весь вывод, переданный драйверу tty, при нажатии клавиши прерывания (прервать, завершить или приостановить). По умолчанию для этой опции принимается значение, заданное для драйвера tty.

keypad
Задает, будут ли функции curses принимать информацию, вводимую с дополнительной клавиатуры терминала. Если эта опция включена, то при нажатии функциональной клавиши (например, клавиши со стрелкой) функция **wgetch** вернет соответствующее ей значение. Если эта опция выключена, то функции curses не будут обрабатывать нажатия функциональных клавиш, и программе потребуется интерпретировать escape-последовательности. Список этих функциональных клавиш приведен в описании функции **wgetch**.

typeahead
Указывает функциям curses, что нужно проверять буферизацию ввода в альтернативном дескрипторе файла.

Сведения о дополнительных опциях curses приведены в описании функций **wgetch** и в разделе Основные сведения о функциях curses для работы с терминалом.

Управление программными метками

Curses содержат функции управления метками функциональных клавиш.

Эти метки появляются в нижней части экрана и упрощают работу с некоторыми приложениями (например, с текстовыми редакторами). Для того чтобы вывести эти метки на экран, необходимо до функций **initscr** или **newterm** вызвать функцию **slk_init**.

Функция	Описание
slk_clear	Удаляет программные метки с экрана.
slk_init	Инициализирует программные метки функциональных клавиш.
slk_label	Возвращает имя текущей метки.
slk_noutrefresh	Обновляет программные метки. Действие этой функции аналогично действию **wnoutrefresh**.
slk_refresh	Обновляет программные метки. Действие этой функции аналогично действию **refresh**.
slk_restore	Отменяет действие функции **slk_clear** и восстанавливает программные метки на экране.
slk_set	Позволяет задать программную метку.
slk_touch	Обновляет программные метки при следующем вызове функции **slk_noutrefresh**.

При работе с программными метками curses уменьшает размер **stdscr** на одну строку. Эта строка резервируется для соответствующих функций управления. При этом значение переменной среды **LINES** также уменьшается. Многие терминалы поддерживают встроенные программные метки. Если такая поддержка включена, curses использует ее. В противном случае программные метки имитируются программно.

На многих терминалах, поддерживающих программные метки, определено 8 меток, этот стандарт принят и для curses. Длина строки метки не превышает 8 символов. Curses допускает два варианта расположения меток: 3-2-3 (3 слева, 2 по центру и 3 справа) или 4-4 (4 слева и 4 справа).

Для того чтобы задать текст метки, вызовите функцию **slk_set**. Эта функция curses также позволяет указать, как следует выровнять метку: по левому краю, по правому краю или по центру. Для получения имени метки без выравнивания вызовите функцию **slk_label**. Функции **slk_clear** и **slk_restore** служат соответственно для удаления и восстановления программных меток. Обычно для обновления программных меток нужно вызвать функцию **slk_noutrefresh** для каждой метки, а затем один раз вызвать функцию **slk_refresh** для получения конечного вывода. Если вы хотите обновить все программные метки при следующем вызове **slk_noutrefresh**, воспользуйтесь функцией **slk_touch**.

Совместимость curses

В этом разделе рассмотрены неполадки, связанные с совместимостью.

Обязательно учтите следующие соображения:

- Библиотека curses в AIX версии 4.3 несовместима с curses AT&T System V версии 3.2.
- Для работы приложений, рассчитанных на curses для AIX версии 4, может потребоваться изменить их исходный код. В curses не применяются расширенные функции AIX curses.
- Приложения с поддержкой многобайтовых символов можно компилировать и компоновать с использованием расширенных функций curses. Использовать их следует только в случае крайней необходимости.

Список дополнительных функций curses

Дополнительные функции curses описаны в следующих разделах:

Дополнительные функции curses описаны в следующих разделах:

- Работа с окнами
- Работа с символами
- Работа с терминалами
- Работа с цветами
- Другие функции

Работа с окнами

Для работы с окнами предназначены следующие функции:

Функция	Описание
scr_dump	Записывает текущее содержимое виртуального экрана в указанный файл
scr_init	Инициализирует структуры данных curses значениями из указанного файла
scr_restore	Выводит на виртуальный экран содержимое текущего файла

Работа с символами

Для работы с символами предназначены следующие функции:

Функция	Описание
echochar, **wechochar** и **pechochar**	Функционально равносильны последовательному вызову функций **addch** (или **waddch**) и **refresh** (или **wrefresh**).
flushinp	Передает в программу все символы из буфера, введенные пользователем, но еще не считанные программой.
insertln или **winsertln**	Выводит в окне пустую строку.
keyname	Возвращает указатель на строку символов, представляющую символьное имя параметра *Key*.
meta	Определяет, может ли функция **wgetch** возвращать однобайтовые символы.
nodelay	Выключает режим блокирующего вызова функции **wgetch**. Если данных для ввода еще нет, функция **wgetch** возвращает **ERR**.
scroll	Прокручивает содержимое окна на одну строку вверх.
unctrl	Возвращает представление символа для печати. Управляющие символы отмечены символом ^.
vwprintw	Выполняет те же действия, что и функция **wprintw**, но поддерживает список аргументов переменной длины.
vwscanw	Выполняет те же действия, что и функция **wscanw**, но поддерживает список аргументов переменной длины.

Работа с терминалами

Для работы с терминалами предназначены следующие функции:

Функция	Описание
def_prog_mode	Устанавливает в качестве текущего режима терминала режим с применением curses
def_shell_mode	Сохраняет в качестве текущего режима терминала режим без применения curses
del_curterm	Освобождает область памяти, на которую указывает переменная *oterm*
notimeout	Запрещает функции **wgetch** устанавливать таймер при обработке входных escape-последовательностей
pechochar	Равносильна последовательному вызову функций **waddch** и **prefresh**.
reset_prog_mode	Возвращает терминал в режим с применением curses.
reset_shell_mode	Возвращает терминал в режим без применения curses (режим оболочки). Функция **endwin** делает это автоматически.
restartterm	Настраивает структуру TERMINAL для работы с curses. Эта функция аналогична функции **setupterm**. Функцию **restartterm** следует вызывать после восстановления предыдущего состояния памяти, например, после вызова функции **scr_restore**.

Работа с цветами

Для работы с цветами предназначены следующие функции:

Функция	Описание
color_content	Возвращает структуру цвета
init_color	Изменяет цвет на указанный
init_pair	Устанавливает в качестве пары цветов указанные цвета фона и текста
pair_content	Возвращает цвет фона и цвет текста, соответствующий заданному номеру

Другие функции

Дополнительно предусмотрены следующие утилиты:

Утилиты	Описание
baudrate	Возвращает скорость вывода данных на текущий терминал
erasechar	Возвращает символ стирания курсором, выбранный пользователем
killchar	Возвращает символ удаления строки, выбранный пользователем

Отладка программ

В AIX предусмотрено несколько отладочных программ: **adb**, **dbx**, **dex**, **softdb**, а также программы отладки ядра. Программа **adb** предназначена для отладки исполняемых двоичных файлов и проверки нетекстовых (не-ASCII) файлов данных.

С помощью программы **dbx** можно отлаживать программы, написанные на языках C, C++ и FORTRAN, на уровне исходного кода, а также исполняемые программы на уровне ассемблера (на машинном уровне). Программа **dex** предоставляет X-интерфейс для программы **dbx**, с помощью которого можно просматривать исходный текст, контекст и переменные прикладной программы. Программа **softdb** схожа с программой **dex**, но **softdb** используется вместе с AIX Software Development Environment Workbench. Программа отладки ядра предназначена для поиска ошибок в коде, запускаемом в ядре.

Подробные сведения о программах отладки **adb** и **dbx** приведены в следующих разделах:

Информация, связанная с данной:

Команда adb

Обзор программы отладки adb

Команда **adb** запускает программу отладки общего назначения. С помощью этой команды вы можете работать с объектными файлами и файлами дампа, а также управлять средой выполнения программ.

Команда **adb** обрабатывает стандартный ввод и направляет данные в стандартный вывод. Клавиши Quit и Interrupt не распознаются. При нажатии этих клавиш команда **adb** ожидает ввода новой команды.

Программа отладки adb - введение

В этом разделе обсуждаются запуск программы **adb** с указанием различных файлов, работа с приглашением **adb**, вызов команд оболочки из программы **adb** и завершение работы **adb**.

Запуск adb с программой

Программу **adb** можно запускать без указания имени файла. В этом случае программа **adb** попытается найти в текущем каталоге файл с именем **a.out** и подготовить его к отладке. Иными словами, команда

```
adb
```

равносильна команде

```
adb a.out
```

Программа **adb** обрабатывает файл **a.out** и ожидает ввода команд. Если файл **a.out** не существует, программа **adb** запускается без обработки файла и не выдает сообщение об ошибке.

Запуск adb с указанием файла дампа

С помощью программы отладки **adb** вы можете анализировать файлы дампа, созданные при возникновении неустранимых системных ошибок. Файлы дампа содержат сведения о содержимом регистров CPU, стека и областей памяти программы на момент возникновения ошибки. Эта информация поможет вам определить причину ошибки.

Для анализа файла дампа и соответствующей программы следует указать одновременно имя файла дампа и имя программы. Команда будет выглядеть следующим образом:

adb *файл-программы файл-дампа*

где *файл-программы* - это имя файла программы, вызвавшей ошибку, а *файл-дампа* - имя созданного системой файла дампа. После ввода команды программа **adb** будет отвечать на вводимые команды, используя информацию из обоих файлов.

Если вы не укажете имя файла дампа, программа **adb** попытается найти в текущем рабочем каталоге файл с именем **core** и открыть его. Если такой файл будет найден, программа **adb** определит, соответствует ли он файлу *файл-программы*. Если да, то он будет применен. В противном случае программа **adb** проигнорирует файл дампа и выдаст сообщение об ошибке.

> **Примечание:** программа **adb** не предназначена для анализа 64-разрядных объектов и файлов дампа в формате AIX 4.3. Программу **adb** можно применять для анализа файлов дампа в формате предыдущих версий AIX 4.3. Однако с помощью smitty систему AIX 4.3 версии 4.3 можно настроить таким образом, чтобы ядро создавало дампы в формате предыдущих версий AIX 4.3.

Запуск adb с файлом данных

Программа **adb** позволяет просматривать содержимое файла в различных форматах и структурах. Для того чтобы проанализировать файл данных с помощью программы **adb**, следует задать его имя вместо файла дампа или файла программы. Например, для просмотра файла с именем **outdata** введите:

```
adb outdata
```

Программа **adb** откроет файл `outdata` и покажет его содержимое. Этот способ эффективен для просмотра файлов с двоичными данными. Если вместо программного файла вы укажете имя файла с двоичными

данными, то команда **adb** может выдать сообщение-предупреждение. Это обычно происходит, если содержимое файла данных похоже на программный файл. Как и файлы дампа, файлы данных не являются исполняемыми.

Запуск adb с опцией записи

Если вы откроете файл с помощью команды **adb -w**, то сможете вносить в него изменения. Пример:

```
adb -w sample
```

открывает файл `sample` для записи. С помощью других команд **adb** вы сможете просматривать и редактировать этот файл. Кроме того, флаг **-w** означает, что если программа **adb** не найдет указанный файл, то она создаст его. Этот флаг также разрешает запись непосредственно в память после выполнения указанной программы.

Приглашение программы adb

После запуска программы **adb** вы можете переопределить приглашение с помощью команды **$P**.

Для замены приглашения **[adb:scat]>>** на строку **Введите команду отладки—–>** необходимо ввести следующую команду:

```
$P"Введите устройству отладки--->"
```

Если вы задаете новое приглашение в командной строке **adb**, кавычки указывать не обязательно.

Вызов команд оболочки из программы adb

Вы можете вызывать команды оболочки, не прерывая работу программы **adb**. Для этого служит команда внешнего вызова **!** (восклицательный знак). Команда внешнего вызова задается в следующем виде:

! *Команда*

Здесь *Команда* - это требуемая команда оболочки. Необходимо указать все обязательные для этой команды аргументы. Программа **adb** передает команду в системную оболочку на выполнение. После завершения команды оболочка возвращает управление программе **adb**. Например, если вы хотите посмотреть дату, введите следующую команду:

```
! date
```

Система покажет дату и вернет управление программе **adb**.

Завершение работы с программой adb

Для завершения работы программы **adb** введите команду **$q** или **$Q**. Эту же функцию выполняет комбинация клавиш Ctrl-D. Клавиши Interrupt и Quit не могут прервать работу программы **adb**. После нажатия этих клавиш программа **adb** будет ожидать ввода новой команды.

Управление выполнением программы

В этом разделе обсуждаются: команды, необходимые для подготовки программ к отладке; выполнение программ; установка, просмотр и удаление точек прерывания; возобновление выполнения программ; пошаговое выполнение; остановка программ и уничтожение процессов.

Подготовка программ к отладке с помощью программы adb

С помощью программы **cc** откомпилируйте программу и поместите ее в файл (например, **adbsamp2**):

```
cc adbsamp2.c -o adbsamp2
```

Для запуска сеанса отладки введите следующую команду:

```
adb adbsamp2
```

Язык программирования C не создает для программ меток операторов, поэтому при работе с программой отладки, обращение к отдельным операторам языка C невозможно. Для более эффективной работы вам следует ознакомиться с тем, какие команды создает компилятор C, и каким образом они связаны с отдельными операторами языка C. Перед тем, как приступать к отладке программы с помощью **adb**, иногда бывает полезно сделать распечатку текста этой программы на ассемблере. Затем, в процессе отладки вы сможете при необходимости обращаться к этой распечатке. Для создания распечатки на языке ассемблера введите команду **cc** с флагом **-S** или **-q***List*.

Например, для создания распечатки программы **adbsamp2.c** введите следующую команду:

```
cc -S adbsamp2.c -o adbsamp2
```

При этом будет создан файл **adbsamp2.s**, содержащий распечатку, а программа будет откомпилирована и помещена в исполняемый файл **adbsamp2**.

Запуск программ

Для запуска программ применяются команды **:r** и **:R**. Формат команд:

[*Адрес*][**,***Счетчик*] **:r** [*Аргументы*]

ИЛИ

[*Адрес*][**,***Счетчик*] **:R** [*Аргументы*]

Здесь параметр *Адрес* указывает адрес, с которого следует начать выполнение программы; параметр *Счетчик* - число точек прерывания, которое следует пропустить; параметр *Аргументы* - аргументы командной строки (например, имена файлов и опции выполнения программы).

Если параметр *Адрес* не указан, **adb** то выполнение программы будет начато с начала. Для запуска программы с начала введите следующую команду:

```
:r
```

Если будет указан параметр *Счетчик*, то программа **adb** проигнорирует указанное число точек прерывания. Например, для пропуска первых пяти точек прерывания нужно ввести следующую команду:

```
,5:r
```

Необходимо отделять аргументы друг от друга по крайней мере одним пробелом. Аргументы передаются программе точно также, как оболочка системы передает программам аргументы командной строки. Это позволяет использовать символы перенаправления оболочки.

Команда **:R** перед началом выполнения программы передает ей аргументы командной строки через оболочку. В аргументах можно применять шаблоны для обозначения множества файлов или других вводимых значений. Перед тем, как передать аргументы программе, оболочка раскрывает такие шаблоны. Эта функция необходима в тех случаях, когда для программы нужно указывать несколько имен файлов. Например, приведенная ниже команда передает аргумент **[a-z]*** оболочке, которая преобразует его в список имен файлов, а затем передает программе:

```
:R [a-z]*.s
```

Команды **:r** и **:R** перед запуском программы удаляют содержимое всех регистров и уничтожают текущий стек. Эта операция останавливает выполнение любой другой копии программы.

Установка точек прерывания

Для создания в программе точек прерывания предусмотрена команда **:b**. Если задана точка прерывания, то по достижении указанного адреса программа останавливает свое выполнение. После этого управление возвращается программе отладки **adb**. Формат команды:

[*Адрес*] [,*Счетчик*] **:b** [*Команда*]

Здесь в параметре *Адрес* должен быть указан допустимый адрес команды; параметр *Счетчик* задает число точек прерывания, которые нужно пропустить, а параметр *Команда* указывает команду **adb**, которую нужно запускать при каждом выполнении команды (вне зависимости от того, останавливается ли программа в точке прерывания). Если указанная команда устанавливает . (точку) рядом со значением 0, то выполнение программы в этой точке будет остановлено.

Мы рекомендуем устанавливать точки прерывания в определенных участках программы, например, в начале выполнения какой-либо функции. Это позволит вам просмотреть содержимое регистров и памяти. Например, для того чтобы при отладке программы **adbsamp2** установить точку прерывания в начале функции **f**, введите следующую команду:

```
.f :b
```

Прерывание произойдет сразу же после передачи управления этой функции, но перед созданием стека функции.

Точку прерывания со счетчиком применяют в функциях, которые вызываются в процессе выполнения программы несколько раз, а также в инструкциях, относящихся к операторам **for** и **while**. Такая точка прерывания позволяет продолжать выполнение программы до тех пор, пока определенная функция или инструкции не будут выполнены заданное число раз. Например, следующая команда устанавливает точку прерывания для функции **f** в программе **adbsamp2** со счетчиком 2:

```
.f,2 :b
```

Таким образом, выполнение программы будет прервано только при втором запуске этой функции.

Просмотр точек прерывания

Определить значение счетчика для любой установленной точки прерывания, а также расположение этой точки можно с помощью команды **$b**. Данная команда выводит список точек прерывания, упорядоченных по адресу, а также все значения счетчиков и команды, определенные для каждой точки. Ниже приведен пример установки двух точек прерывания в файле **adbsamp2** и их просмотра с помощью команды **$b**:

```
.f+4:b
.f+8:b$v
$b
точки прерывания
count  brkpt            command
1      .f+8             $v
1      .f+4
```

После запуска программа остановится на первой же точке прерывания, например, `.f+4`. Если вы продолжите выполнение программы с помощью команды **:c**, она вновь остановится на следующей точке прерывания, после чего будет выполнена команда **$v**. Вот как будет выглядеть соответствующая последовательность команд и ответов:

```
:r
adbsamp2:running
breakpoint        .f+4:         st       r3,32(r1)
:c
adbsamp2:running
variables
b = 268435456
```

```
d = 236
e = 268435512
m = 264
breakpoint        .f+8          l       r15,32(r1)
```

Удаление точек прерывания

Формат команды **:d** (удаление точки прерывания из программы):

Адрес **:d**

Здесь параметр *Адрес* задает адрес удаляемой точки прерывания.

Например, для удаления точки прерывания, соответствующей началу функции **f** в программе **adbsamp2**, введите следующую команду:

```
.f:d
```

Возобновление выполнения программ

Формат команды **:c** (продолжение выполнения программы после остановки в точке прерывания):

[*Адрес*] [,*Счетчик*] **:c** [*Сигнал*]

Здесь параметр *Адрес* указывает адрес, с которого следует возобновить выполнение программы; параметр *Счетчик* - число точек прерывания, которое нужно пропустить; параметр *Сигнал* - номер сигнала, который нужно направить программе.

Если параметр *Адрес* не указан, то программа возобновит выполнение с первой же команды, следующей за точкой прерывания. Если указать значение параметре *Счетчик*, то программа отладки **adb** проигнорирует соответствующее число точек прерывания.

Если выполнение программы прервано с помощью клавиши Interrupt или Quit, то этот сигнал будет автоматически передан программе при ее перезапуске. Для того чтобы не передавать сигнал в таких случаях, введите следующую команду:

[*Адрес*] [,*Счетчик*] **:c 0**

Аргумент **0** запрещает отправку сигнала подпроцессу.

Выполнение отдельной инструкции программы

Для выполнения отдельной инструкции или пошагового выполнения программы предусмотрена команда **:s**. Эта команда выполняет инструкцию и возвращает управление программе отладки **adb**. Формат команды:

[*Адрес*] [,*Счетчик*] **:s** [*Сигнал*]

Здесь параметр *Адрес* указывает адрес инструкции, которую нужно выполнить, а *Счетчик* - число выполняемых команд. Если в настоящее время никакие подпроцессы не выполняются, то в качестве подпроцесса будет запущен указанный *Объектный_файл*. В этом случае передача сигнала невозможна и остальная часть строки будет рассматриваться как аргументы подпроцесса. Если не указать значение параметра *Адрес*, то программа **adb** использует текущий адрес. Если определить параметр *Счетчик*, то программа **adb** последовательно запустит соответствующее число команд. В режиме пошагового выполнения точки прерывания игнорируются. В качестве примера, ниже приведена команда для запуска первых пяти команд функции **main**:

```
.main,5:s
```

Остановка выполнения программ с помощью клавиш interrupt и quit

Остановить выполнение программы можно с помощью клавиш Interrupt и Quit. При нажатии любой из этих клавиш текущая программа будет остановлена, а управление будет передано программе отладки **adb**. Эти клавиши оказываются полезными при отладке программ с бесконечными циклами и другими ошибками.

При нажатии клавиши Interrupt или Quit программа **adb** автоматически сохраняет сигнал. При возобновлении выполнения программы с помощью команды **:c**, программа отладки **adb** автоматически направляет сигнал программе. Эта функция полезна при тестировании программ, использующих сигналы в процессе выполнения. Для того чтобы возобновить выполнение программы, не направляя при этом сигналов, введите следующую команду:

```
:c 0
```

Аргумент **0** запрещает отправку сигнала программе.

Остановка выполнения программы

Для остановки отлаживаемой программы применяется команда **:k**. Эта команда останавливает процесс программы и возвращает управление программе отладки **adb**. При этом содержимое системных регистров и стека очищается, а программа перезапускается. Ниже показан пример применения команды **:k** для удаления текущего процесса из программы **adb**:

```
:k
560:    killed
```

Применение выражений в программе adb

Этот раздел посвящен применению выражений в **adb**.

Выражения с целыми числами

В выражении можно указывать целые числа в десятичном, восьмеричном и шестнадцатеричном формате. Десятичное целое должно начинаться с десятичной цифры, отличной от нуля. Восьмеричное число должно начинаться с нуля (0) и содержать цифры от 0 до 7. Шестнадцатеричное число должно начинаться с префикса 0x и может содержать десятичные цифры и буквы от a до f (как в нижнем, так и в верхнем регистре). Ниже приведены примеры правильной записи чисел:

Десятичные	Восьмеричные	Шестнадцатеричные
34	042	0x22
4090	07772	0xffa

Выражения с идентификаторами

Идентификаторы - это имена глобальных переменных и функций, определенных в отлаживаемой программе. Идентификаторы эквивалентны адресу данной переменной или функции. Они хранятся в таблице символьных имен (идентификаторов) программы и доступны, если эта таблица не удалена из программного файла.

В выражениях идентификатор может записываться в том виде, в каком он был задан в исходной программе, или в том виде, в каком он хранится в таблице символьных имен. Длина идентификатора в таблице не должна превышать 8 символов.

Если вы указываете команду **?**, программа **adb** создает символьные адреса на основе имен из таблицы. В результате в выводе команды **?** иногда встречаются имена функций. Этого не происходит, если команда **?** используется для текста (команд), а команда **/** используется для данных.

Вы сможете обратиться к локальной переменной только в том случае, если при компиляции исходной программы на языке C был указан флаг **-g**.

Если флаг **-g** не был задан, адрес локальной переменной определить нельзя. Получить значение локальной переменной **b** функции sample можно с помощью следующей команды:

```
.sample.b / x - значение локальной переменной.
.sample.b = x - адрес локальной переменной.
```

Выражения с операторами

Целые числа, идентификаторы, переменные и имена регистров можно использовать в выражениях со следующими операторами:

Унарные операторы:

~ (тильда)	Побитовое дополнение
- (тире)	Целое отрицание
* (звездочка)	Возвращает содержимое ячейки с указанным адресом

Бинарные операторы:

+ (плюс)	Сложение
- (минус)	Вычитание
* (звездочка)	Умножение
% (процент)	Целочисленное деление
&(амперсанд)	Побитовое умножение
] (правая квадратная скобка)	Побитовое сложение
^ (символ вставки)	Модуль
# (номер)	Округление до ближайшего большего кратного числа

В программе **adb** используется 32-разрядная арифметика. Значения, превышающие десятичное число 2 147 483 647, представляются отрицательными величинами. В следующем примере показан результат присваивания двух разных значений переменной *n*, и вывод ее значения в десятичном и шестнадцатеричном виде:

```
2147483647>n<
n=D
    2147483647<
n=X
    7fffffff
2147483648>n<
n=D
    -2147483648<
n=X
    80000000
```

Унарные операторы имеют более высокий приоритет, чем бинарные. Все бинарные операторы имеют одинаковый приоритет и вычисляются слева направо. Ниже приведены примеры вычисления выражений программой **adb**:

```
2*3+4=d
    10
4+2*3=d
    18
```

Для изменения приоритета операций применяются круглые скобки. Например, если в предыдущем выражении использовать скобки, то его значение изменится следующим образом:

```
4+(2*3)=d
    10
```

Унарный оператор * (звездочка) рассматривает заданный адрес как указатель на сегмент данных. Значение выражения, в котором используется этот оператор, равно значению, на которое указывает этот указатель. Например, значение выражения

```
*0x1234
```

равно величине, которая хранится в ячейке с адресом 0x1234, тогда как значение
`0x1234`

равно 0x1234.

Настройка программы отладки adb

В этом разделе описана настройка программы **adb**.

Объединение нескольких команд в одной строке

В одной строке можно указывать несколько команд, отделяя их друг от друга точкой с запятой (;). Команды выполняются поочередно слева направо. Изменения в текущем адресе и формате передаются следующей команде. Если при выполнении какой-либо команды происходит ошибка, остальные команды игнорируются. Например, следующая последовательность команд перечисляет переменные **adb** и активные функции в одной из точек программы программы **adbsamp2**:

```
$v;$c
variables
b = 10000000
d = ec
e = 10000038
m = 108
t = 2f8.
f(0,0) .main+26.
main(0,0,0) start+fa
```

Создание сценариев adb

Если при запуске программы **adb** перенаправить стандартный файл ввода, то **adb** сможет считывать команды не с клавиатуры, а из текстового файла. Для этого введите символ переправления < (знак "меньше") и укажите имя файла. Например, для того чтобы команды считывались из файла script, введите:

```
adb sample <script
```

В этом файле должны быть указаны допустимые команды **adb**. В случаях, когда один и тот же набор команд применяется для разных объектных файлов, воспользуйтесь файлами сценариев программы **adb**. С помощью сценариев можно выводить на экран содержимое файлов дампа после того, как в программе произошла ошибка. Файл, содержащий команды выдачи информации о программной ошибке, показан в следующем примере. Если сделать его входным файлом для программы **adb**, использующей для отладки файла **adbsamp2** указанную ниже команду, то вывод будет выглядеть следующим образом:

```
120$w
4095$s.
f:b:
r
=1n"======= Переменные adb ======="
$v
=1n"======= Таблица адресов ======="
$m
=1n"======= Обратная трассировка стека C ======="
$C
=1n"======= Внешние переменные C ======="
$e
=1n"======= Регистры ======="
$r
0$s
=1n"======= Сегмент данных ======="<
b,10/8xna
```

```
$ adb adbsamp2 <script
```

```
adbsamp2:running
breakpoint .f:   b  .f+24
    ======= Переменные adb =======
```

```
variables
0 = TBD
1 = TBD
2 = TBD
9 = TBD
b = 10000000
d = ec
e = 10000038
m = 108
t = 2f8
    ======= Таблица адресов =======
[0]? map  .adbsamp2.
b1 = 10000000  e1 = 100002f8  f1 = 0
b2 = 200002f8  e2 = 200003e4  f2 = 2f8
[0]/ map  .-.
b1 = 0    e1 = 0    f1 = 0
b2 = 0    e2 = 0    f2 = 0
    ======= Обратная трассировка стека C =======.
f(0,0) .main+26.
main(0,0,0) start+fa
    ======= Внешние переменные C =======Полное слово.
errno: 0.
environ:  3fffe6bc.
NLinit:  10000238.
main: 100001ea.
exit: 1000028c.
fcnt: 0

.loop .count:  1.
f:   100001b4.
NLgetfile: 10000280.
write: 100002e0.
NLinit. .X:  10000238 .
NLgetfile. .X:   10000280 .
cleanup: 100002bc.
exit: 100002c8 .
exit . .X:  1000028c . .
cleanup . .X:  100002bc

    ======= Регистры =======
mq  20003a24  .errno+3634
cs  100000 gt
ics  1000004
pc  100001b4 .f
r15  10000210 .main+26
r14  20000388  .main
r13  200003ec  .loop .count
r12  3fffe3d0
r11  3fffe44c
r10  0
r9  20004bcc
r8  200041d8  .errno+3de8
r7  0
r6  200030bc  .errno+2ccc
r5  1
r4  200003ec  .loop .count
r3  f4240
r2  1
r1  3fffe678
r0  20000380  .f.
f:   b  .f+24

    ======= Сегмент данных =======
10000000:  103 5313  3800  0  0  2f8 0  ec
10000010:  0  10 1000  38 0  0  0  1f0
10000020:  0  0  0  0  1000  0  2000  2f8
10000030:  0  0  0  0  4  6000  0  6000
10000040:  6e10  61d0  9430  a67 6730  6820  c82e  8
10000050:  8df0  94 cd0e  60 6520  a424  a432  c84e
```

```
10000060:  8  8df0  77 cd0e  64 6270  8df0  86
10000070:  cd0e  60 6520  a424  a432  6470  8df0  6a
10000080:  cd0e  64 c82e  19 8df0  78 cd0e  60
10000090:  6520  a424  a432  c84e  19 8df0  5b cd0e
100000a0:  64 cd2e  5c 7022  d408  64 911 c82e
100000b0:  2e 8df0  63 cd0e  60 6520  a424  a432
100000c0:  c84e  2e 8df0  46 cd0e  64 15 6280
100000d0:  8df0  60 cd0e  68 c82e  3f 8df0  4e
100000e0:  cd0e  60 6520  a424  a432  c84e  3f 8df0
100000f0:  31 cd0e  64 c820  14 8df0  2b cd0e
10000100:
```

Установка ширины вывода

Максимальная ширина строки (в символах), выдаваемой программой **adb**, задается с помощью команды **$w**. Формат команды:

*Ширина***$w**

В этом формате параметр *Ширина* - целое число, задающее длину выдаваемой на дисплей строки в символах. Можно задавать любую ширину, подходящую для вашего дисплея. При первом вызове программы **adb** устанавливается ширина по умолчанию, равная 80 символам.

Данная команда может использоваться при перенаправлении вывода на построчный принтер или на специальное устройство вывода. Например, ширина строки дисплея, равная 120 символам (обычно это максимальная длина строки для построчных принтеров), устанавливается с помощью команды:

```
120$w
```

Установка максимального смещения

Программа **adb** обычно выдает адреса памяти и файлов в виде суммы идентификатора и смещения. Такой формат позволяет установить связь между командами и данными, показанными на экране, и конкретной функцией или переменной. При запуске программа **adb** устанавливает максимальное смещение, равное 255, так что символьные адреса присваиваются только тем командам и данным, которые отстоят от начала функции или переменной меньше чем на 256 байтов. Остальные команды и данные получают числовые адреса.

Во многих программах фактический размер функции или переменной превышает 255 байт. По этой причине для работы с большими программами в **adb** предусмотрена возможность изменения максимального смещения. Для этого предназначена команда **$s**.

Формат команды:

*Смещение***$s**

Параметр *Смещение* - целое число, задающее новое значение смещения. Например, следующая команда увеличивает максимально возможное смещение до 4095:

```
4095$s
```

После выполнения этой команды всем командам и данным размером меньше 4096 байт будут присваиваться символьные адреса. Можно отключить режим присваивания символьных адресов, указав нулевое смещение. В этом случае все адреса будут числовыми.

Установка формата данных ввода по умолчанию

Для изменения формата чисел по умолчанию применяются команды **$d** и **$o** (octal - восьмеричный). Формат по умолчанию определяет способ интерпретации в программе **adb** чисел, не начинающихся с 0

(восьмеричные) или с 0x (шестнадцатеричные), и представление чисел на экране, если формат не указан. Эти команды можно использовать для работы с комбинациями десятичных, восьмеричных и шестнадцатеричных чисел.

Команда **$o** задает переход к восьмеричной системе счисления. После ввода этой команды программа **adb** будет выдавать все числа в восьмеричном формате, за исключением тех, для которых указан другой формат.

Формат команды **$d** следующий: *Основание***$d**, где *Основание* - новое основание системы счисления. Если параметр *Основание* не указан, то команда **$d** устанавливает значение по умолчанию, равное 16. При первом запуске программы **adb** устанавливается шестнадцатеричный формат, который будет использоваться по умолчанию. Если вы изменили формат по умолчанию, то восстановить прежнее значение можно с помощью команды **$d** без параметра:

```
$d
```

Для того чтобы установить десятичный формат, введите следующую команду:

```
0xa$d
```

Изменение режима дезассемблирования

С помощью команд **$i** и **$n** можно задать набор команд и мнемонику, которые должны применяться программой **adb** для дезассемблирования команд. Команда **$i** определяет набор команд, применяемых для дезассемблирования; команда **$n** - мнемонику, используемую при дезассемблировании.

Если эти команды введены без параметров, то будут показаны текущие значения.

Возможные параметры команды **$i**:

com
Задает набор команд, общих для архитектур PowerPC и семейство POWER.

pwr
Задает набор команд и мнемонику для реализации POWER архитектуры POWER Architecture.

pwrx
Задает набор команд и мнемонику для реализации POWER2 архитектуры семейство POWER.

ppc
Задает набор команд и мнемонику для PowerPC.

601
Задает набор команд и мнемонику для микропроцессор RISC PowerPC 601.

603
Задает набор команд и мнемонику для микропроцессор RISC PowerPC 603.

604
Задает набор команд и мнемонику для микропроцессор RISC PowerPC 604.

ANY
Определяет любую допустимую команду. Для перекрывающихся наборов команд по умолчанию будет установлена мнемоника PowerPC.

Возможные параметры команды **$n**:

pwr
Задает набор команд и мнемонику для реализации POWER архитектуры POWER Architecture.

ppc
Задает мнемонику для реализации архитектуры PowerPC.

Арифметические выражения и вывод текста

Команда **=** (знак равенства) позволяет выполнять в программе **adb** арифметические вычисления. В ответ на эту команду программа **adb** покажет значение выражения в определенном формате.

Эта команда может преобразовывать числа из одного формата в другой, проверять арифметические вычисления в программе отладки и показывать сложные адреса в упрощенном виде. Ниже приведена команда преобразования шестнадцатеричного числа 0x2a в десятичное:

```
0x2a=d
        42
```

Аналогично, следующая команда преобразует значение 0x2a в символ ASCII * (звездочка):

```
0x2a=c
        *
```

Выражения в команде могут состоять из любых комбинаций символов и операторов. Например, следующая команда вычисляет значение на основе содержимого регистров **r0** и **r1** и переменной **b** программы **adb**.

```
<r0-12*<r1+<b+5=X
    8fa86f95
```

Кроме того, вы можете определить значение внешнего символа и проверить шестнадцатеричное значение адреса внешнего символа:

```
main+5=X
    2000038d
```

Команда **=** (знак равенства) может также применяться для просмотра литеральных строк. Эта функция позволяет просматривать в программе **adb** комментарии к выполняемому сценарию. Например, следующая команда вводит три пустых строки и затем печатает сообщение `C Stack Backtrace`:

```
=3n"C Stack Backtrace"
```

Просмотр и редактирование исходного файла с помощью программы adb

Этот раздел посвящен просмотру и редактированию исходных файлов с помощью программы **adb**.

Просмотр команд и данных

В программе **adb** предусмотрено несколько команд, позволяющих выводить на экран команды и данные из конкретной программы, а также данные из указанного файла. Они перечислены ниже:

Показать адрес
Адрес [, *Число*] **=***Формат*

Показать команду
Адрес [, *Число*] **?***Формат*

Показать значение переменной
Адрес [, *Число*] **/** *Формат*

В этом формате символы и переменные имеют следующее значение:

Адрес
Расположение команды или элемента данных.

Число
Количество выдаваемых элементов.

Формат
Вид элемента на экране.

= Выдает адрес элемента.

? Выдает команду в текстовом сегменте.

/ Выдает значение переменной.

Формирование адресов

В программе **adb** адреса представляют собой 32-разрядные значения, указывающие на конкретные адреса памяти. Адреса могут быть представлены в следующем виде:

Абсолютный адрес
32-разрядная величина, представляемая 8-значным шестнадцатеричным числом или его эквивалентом в другой системе счисления.

Имя символа
Расположение (адрес) функции или переменной, определенной в программе, может быть представлено ее именем (идентификатором) в программе.

Точки ввода
Точка входа в процедуру представляется именем процедуры, перед которым стоит . (точка). Например, ссылка на адрес начала функции `main` обозначается следующим образом:

```
.main
```

Смещение
Для обозначения адресов других точек в программе можно указывать их смещение относительно точки входа в программу. Например, запись адреса команды, отстоящей на 4 байта от точки входа в `main`, выглядит следующим образом:

```
.main+4
```

Просмотр адресов

Для просмотра адресов в заданном формате предназначена команда **=** (знак равенства). С помощью этой команды можно просматривать адреса команд и данных (в более простой форме), а также результаты вычисления арифметических выражений. Например, команда

```
main=an
```

выдает адрес функции `main`:

```
10000370:
```

В следующем примере приведена команда, которая выдает десятичное значение суммы внутренней переменной **b** и шестнадцатеричного числа 0x2000:

```
<b+0x2000=D
    268443648
```

Если задано число, то одно и то же значение повторяется указанное число раз. Следующий пример содержит команду, с помощью которой дважды выводится значение функции `main`, а затем результат выполнения этой функции:

```
main,2=x
    370 370
```

Если адрес не указан, то используется текущий адрес. После однократного выполнения указанной выше команды (устанавливающей адрес `main` в качестве текущего), следующая команда выполняет то же действие:

```
,2=x
    370 370
```

Если формат не указан, то программа **adb** будет использовать последний применявшийся формат. Например, если вводится следующая последовательность команд, то и `main`, и `one` выдаются в шестнадцатеричном формате:

```
main=x
  370
one=
  33c
```

Просмотр данных обратной трассировки стека C

Для трассировки всех активных функций предназначена команда **$c**. Эта команда показывает список всех вызванных функций, которые еще не вернули управление вызвавшей их процедуре. Кроме того, она выдает список адресов, из которых были вызваны функции, и список переданных в них аргументов. Например, следующая последовательность команд задает точку прерывания по адресу функции `.f+2` в программе **adbsamp2**. Точка прерывания вызывает команду **$c**. Программа запускается, доходит до точки прерывания, а затем выдает данные обратной трассировки вызванных функций C:

```
.f+2:b$c
:r
adbsamp2:running
.f(0,0) .main+26
.main(0,0,0) start+fa
breakpoint                f+2:           tgte     r2,r2
```

По умолчанию команда **$c** показывает все вызовы. Если требуется только часть вызовов, укажите нужное число вызовов. Например, следующая команда будет показывать только одну активную функцию в предыдущей точке прерывания:

```
,1$c
.f(0,0) .main+26
```

Выбор форматов данных

Формат - это буква или символ, определяющие способ представления данных на экране. Ниже перечислены наиболее распространенные форматы:

a Текущий символьный адрес

b Один байт в восьмеричном представлении (показывает данные, связанные с командами, либо старший или младший байт регистра)

c Один байт в символьном виде (переменные типа char)

d Полуслово в десятичном представлении (короткий тип)

Д Полное слово в десятичном представлении (длинный тип)

i Машинные команды в мнемоническом формате

n Новая строка

o Полуслово в восьмеричном представлении (короткий тип)

0 Полное слово в восьмеричном представлении (длинный тип)

r Пробел

s Строка символов, заканчивающаяся символом NULL (массивы переменных типа char, заканчивающиеся символом NULL)

t Горизонтальная табуляция

u Полуслово как целое без знака (короткий тип)

x Полуслово в шестнадцатеричном представлении (короткий тип)

X Полное слово в шестнадцатеричном представлении (длинный тип)

Ниже приведены результаты выполнения некоторых команд, применявшихся с программой **adbsamp**:

```
main=o
    1560

main=O
    4000001560

main=d
    880

main=D
    536871792

main=x
    370

main=X
    20000370

main=u
    880
```

Можно указывать как один, так и несколько форматов. Для того чтобы сделать данные на экране более удобными для чтения, вы можете указывать форматы **a**, **n**, **r** и **t** в сочетании с другими форматами.

Изменение карты распределения памяти

С помощью команд **?m** и **/m** можно изменять значения в карте распределения памяти. Команды присваивают указанные значения соответствующим записям карты. Формат команд:

```
[,число] ?m b1 e1 f1
[,число] /m b1 e1 f2
```

В следующем примере показан результат применения этих команд к схеме распределения памяти, выданной командой **$m** в предыдущем примере:

```
,0?m    10000100         10000470       0
/m      100      100     100
$m
  [0] :   ?map :   'adbsamp3'
 b1 = 0x10000100,  e1 = 10000470,  f1 = 0
 b2 = 0x20000600,  e2 = 0x2002c8a4, f2 = 0x600

 [1] :  ?map :   'shr.o' в библиотеке '/usr/ccs/lib/libc.a'
 b1 = 0xd00d6200,  e1 = 0xd01397bf,  f1 = 0xd00defbc
 b2 = 0x20000600,  e2 = 0x2002beb8, f2 = 0x4a36c

  [-] :  /map :          '-'
 b1 = 100,    e1 = 100,   f1 = 100
 b2 = 0,    e2 = 0,     f2 = 0
```

Для того чтобы изменить значения сегментов данных, добавьте после / или **?** звездочку (*).

```
,0?*m   20000270         20000374       270
/*m     200      200     200
$m
  [0] : ?map :   'adbsamp3'
 b1 = 0x10000100,  e1 = 10000470,  f1 = 0
 b2 = 0x20000270,  e2 = 0x20000374, f2 = 0x270

 [1] :  ?map :   'shr.o' в библиотеке '/usr/ccs/lib/libc.a'
 b1 = 0xd00d6200,  e1 = 0xd01397bf,  f1 = 0xd00defbc
 b2 = 0x20000600,  e2 = 0x2002beb8, f2 = 0x4a36c
```

```
 [-] :  /map :          '-'
b1 = 100,    e1 = 100,   f1 = 100
b2 = 0,    e2 = 0,     f2 = 0
```

Корректировка двоичных файлов

Запустив программу **adb** с флагом **-w** и воспользовавшись командами **w** и **W** ( ), можно корректировать и изменять любые файлы, включая исполняемые двоичные файлы.

Поиск значений в файле

Для поиска значений в файле служат команды **l** и **L**. Формат команд:

?l *Значение*

ИЛИ

/l*Значение*

Программа начинает поиск с текущего адреса и ищет указанное *Значение*. Команда **l** предназначена для поиска 2-байтовых значений, команда **L** - 4-байтовых.

Команда **?l** начинает поиск с текущего адреса и продолжает его до первого совпадения или до конца файла. Если значение найдено, то его адрес становится текущим адресом. Например, следующая команда ищет первое вхождение символа **f** в файле **adbsamp2**:

```
?l .f.
write+a2
```

В данном примере значение найдено по адресу **.write+a2**, который стал текущим.

Запись в файл

Для внесения изменений в файл служат команды **w** и **W**. Формат команд:

[*Адрес*] **?w***Значение*

Здесь *Адрес* - это адрес значения, которое вы хотите изменить, а *Значение* - новое значение. Команда **w** предназначена для записи 2-байтовых значений, команда **W** - 4-байтовых. Например, следующие команды изменяют слово "This" на "The":

```
?l .Th.
?W .The.
```

Команда **W** изменяет все четыре символа.

Корректировка значений в памяти

Вы можете вносить изменения в программу даже во время ее выполнения. Если для запуска программы вы использовали команду **:r** с точкой прерывания, то последующее выполнение команд **w** приведет к тому, что программа **adb** будет корректировать не файл, а программу в памяти. Эта команда предназначена для внесения изменений в данные программы во время ее выполнения, например, для временной корректировки флагов и переменных.

Применение переменных программы adb

При запуске программа **adb** автоматически создает набор своих собственных переменных. Эти переменные содержат адреса и размеры различных частей программного файла (см. приведенную ниже таблицу):

0 Последнее напечатанное значение

1 Последний блок смещения в источнике инструкций

2 Предыдущее значение переменной **1**

9 Счетчик в последней команде **$<** или **$<<**.

b Базовый адрес сегмента данных

d Размер сегмента данных

e Адрес входа программы

m Сигнатура

s Размер сегмента стека

t Размер текстового сегмента

Программа **adb** читает программный файл и ищет значения этих переменных. Если обнаруживается, что файл не программный, то **adb** оставляет значения неопределенными.

Для просмотра значений, присваиваемых программой **adb** этим переменным, служит команда **$v** . Эта команда выдает список имен переменных и их значений в текущем формате. В списке содержатся только те переменные, значение которых не равно нулю (0). Кроме того, если переменная имеет ненулевое значение сегмента, то ее значение выдается как адрес, иначе - как число. В следующем примере показаны результаты применения этой команды к программе **adbsamp**:

```
$v
Variables
0 = undefined
1 = undefined
2 = undefined
9 = undefined
b = 10000000
d = 130
e = 10000038
m = 108
t = 298
```

Текущее значение переменной программы **adb** можно указать в выражении, задав имя переменной со знаком < (меньше) перед ним. В следующем примере выдается текущее значение переменной **b**:

```
<b=X
10000000
```

Вы можете создавать свои собственные переменные или изменять значения существующих переменных, присваивая им значения с помощью знака > (больше). Формат:

выражение > *имя-переменной*

где *Выражение* - это значение, присваиваемое переменной, а *имя-переменной* - это переменная, которой присваивается значение. Параметр *Имя_переменной* должен состоять из одной буквы. Например, команда

```
0x2000>b
```

присваивает шестнадцатеричное значение 0x2000 переменной **b**. Для того чтобы убедиться, что присвоение выполнено, можно просмотреть содержимое **b**:

```
<b=X
 2000
```

Поиск текущего адреса

В программе **adb** есть две переменные, которые отслеживают последний адрес, применявшийся командой, и последний адрес, введенный в команде. Переменная **.** (точка) содержит последний адрес, применявшийся командой (он также называется текущим адресом). Последний адрес, введенный в команде, хранится в переменной **"** (двойная кавычка). Переменная **.** и **"** обычно содержат один и тот же адрес, за исключением случаев применения таких неявных команд, как символ новой строки и символ вставки (^). Эти символы автоматически увеличивают и уменьшают значение переменной **.** , но оставляют неизменной значение переменной " .

Обе переменные, **.** и **"**, могут использоваться в любых выражениях. Указывать знак < (меньше) не требуется. Ниже приведен пример команд для выдачи значений этих переменных в начале отладки программы **adbsamp**.

```
.=
    0.
=
    0
```

Просмотр внешних переменных

Для просмотра значений всех внешних переменных в программе **adb** предназначена команда **$e**. Внешними называются переменные пользовательской программы, имеющие глобальную область действия или определенные вне всех функций, а также переменные, определенные в библиотечных процедурах, используемых программой, и все внешние переменные общих библиотек.

Команда **$e** позволяет получить список всех имеющихся переменных и их значений. В каждой строке вывода команды указывается имя переменной и его значение (если оно есть). Если команда вводится с параметром *Число*, то печатаются только внешние переменные, связанные с этим файлом.

В следующем примере иллюстрируется установка точки прерывания, вызывающей команду **$e**, и приводятся результаты выполнения этой команды при запуске программы **adbsamp2** (не забудьте удалить другие точки прерывания, которые могли быть установлены ранее):

```
.f+2:b,0$e
:r
adbsamp2:running
_errno: 0
_environ:  3fffe6bc
__NLinit:  10000238
_main: 100001ea
_exit: 1000028c
_fcnt: 0
_loop_count:  1
_f:   100001b4
_NLgetfile: 10000280
_write: 100002e0
__NLinit__X:  10000238
_NLgetfile__X:   10000280
__cleanup: 100002bc
__exit: 100002c8
_exit__X:  1000028c
__cleanup__X:  100002bc
breakpoint .f+2:  st r2,1c(r1)
```

Просмотр таблиц адресов

Программа **adb** подготавливает набор таблиц для сегментов текста и данных, с помощью которых она обращается к элементам, выводимым на экран по вашему запросу. Для просмотра содержимого этих таблиц адресов служит команда **$m**. Команда выдает таблицы для всех сегментов программы, используя информацию либо из программного файла и файла дампа, либо непосредственно из памяти.

Команда **$m** выдает примерно следующую информацию:

```
$m
  [0] : ?map :   'adbsamp3'
  b1 = 0x10000200,  e1 = 0x10001839,  f1 = 0x10000200
  b2 = 0x2002c604,  e2 = 0x2002c8a4,  f2 = 0x600

  [1] : ?map :   'shr.o' в библиотеке 'lib/libc.a'
  b1 = 0xd00d6200,  e1 = 0xd013976f,  f1 = 0xd00defbc
  b2 = 0x20000600,  e2 = 0x2002bcb8,  f2 = 0x4a36c

  [-] :  /map :          '-'
  b1 = 0x0000000,  e1 = 0x00000000,  f1 = 0x00000000
  b2 = 0x0000000,  e2 = 0x00000000,  f2 = 0x00000000
```

Здесь показаны параметры распределения адресов памяти для сегментов текста (`b1`, `e1` и `f1`) и данных (`b2`, `e2` и `f2`) для двух файлов, применяемых программой **adb**. Данные значения относятся только к приведенному примеру программы **adbsamp3**. Второй набор значений таблицы соответствует используемому файлу дампа. Так как на самом деле никакой файл дампа не используется, вместо имени файла указан символ **-** (тире).

Значение в квадратных скобках может использоваться в качестве параметра *Число* в командах **?e** и **?m**.

Справочная информация о программе отладки adb

Для организации и управления данными программа отладки **adb** использует адреса, выражения, операторы, команды и переменные.

Адреса в программе отладки adb

Адрес в файле, соответствующий указанному адресу, определяется по схеме, связанной с выбранным файлом. Каждая схема представлена двумя тройками (*B1*, *E1*, *F1*) и (*B2*,*E2*, *F2*). Параметр *Адрес_в_файле*, соответствующий указанному параметру *Адрес*, вычисляется следующим образом:

B1<=Адрес<E1=>Адрес_в_файле=Адрес+F1-B1

ИЛИ

B2<=Адрес<E2=>Адрес_в_файле=Адрес+F2-B2

Параметр *Адрес* должен лежать в диапазоне *B1 - E1* или *B2 - E2*. В некоторых случаях, например, в программах с разделенными областями I и D, два сегмента для одного файла могут перекрываться. Если после команды **?** (вопросительный знак) или **/** (косая черта) указан символ ***** (звездочка), то применяется только вторая тройка.

Начальный вид обеих схем может применяться для стандартных файлов **a.out** и **core**. Если какой-либо из этих файлов заменен на другой, то для него параметры *B1* и *F1* будут равны 0, а для параметра *E1* будет задан максимальный размер файла. В этом случае весь файл может быть просмотрен без преобразования адресов.

Выражения в программе отладки adb

Программа отладки **adb** поддерживает следующие выражения:

. (точка)
Указывает последний адрес, заданный для команды. Последний адрес также называется текущим адресом.

+ (сложение)
Увеличивает значение . (точку) на текущее приращение.

^ (символ вставки)
Уменьшает значение . (точку) на текущее приращение.

" (двойные кавычки)
Указывает последний адрес, напечатанный командой.

Целочисленный
Задает восьмеричное число, если начинается с **0o**, шестнадцатеричное - если с **0x** или **#**, и десятичное - если с **0t**. В остальных случаях это выражение задает число, интерпретируемое в текущей системе счисления. Первоначально основание системы счисления равно 16.

` Cccc '
Задает значение ASCII длиной до 4 символов. Вместо символа ' (апостроф) может применяться символ \ (обратная косая черта).

<Имя
Считывает текущее значение параметра *Имя*. Параметр *Имя* задает имя переменной или имя регистра. Команда **adb** обрабатывает переменные, имена которых состоят из одной буквы или цифры. Если параметр *Имя* задает имя регистра, то значение в этом регистре определяется из системного заголовка параметра *Файл_дампа*. Вы можете просмотреть допустимые имена регистров с помощью команды **$r**.

Символ
Задает последовательность прописных и строчных букв, знаков подчеркивания или цифр, причем первым символом не может быть цифра. Значение параметра *Символ* определяется из таблицы символов в параметре *Объектный_файл*. При необходимости параметр *Символ* может начинаться с символа _ (знак подчеркивания).

. Символ
Задает точку входа функции, указанной в параметре *Символ*.

Процедура.Имя
Задает адрес параметра *Имя* в указанной процедуре на языке C. *Процедура* и *Имя* являются параметрами типа *Символ*. Если параметр *Имя* пропущен, то это значение задает адрес активизированного последним кадра стека C, соответствующего параметру *Процедура*.

(выражение)
Задает значение выражения.

Операторы в программе отладки adb

Целые числа, символы, переменные и имена регистров можно комбинировать в выражениях с помощью следующих операторов:

Унарные операторы

***** *Выражение*
Возвращает содержимое расположения, заданного параметром *Выражение*, в параметре *Файл_дампа*.

@ *Выражение*
Возвращает содержимое расположения, заданного параметром *Выражение*, в параметре *Объектный_файл*.

- *выражение*
Изменяет знак целочисленного выражения.

~ *Выражение*
Выполняет поразрядное отрицание.

*Выражение*
Выполняет логическое отрицание.

Бинарные операторы

Выражение1+*Выражение2*
Выполняет целочисленное сложение.

выражение-1-*выражение-2*
Выполняет целочисленное вычитание.

*Выражение1***Выражение2*
Выполняет целочисленное умножение.

Выражение1%*Выражение2*
Выполняет целочисленное деление.

Выражение1&*Выражение2*
Выполняет поразрядную конъюнкцию.

Выражение1|*Выражение2*
Выполняет поразрядную дизъюнкцию.

Выражение1#*Выражение2*
Округляет параметр *Выражение1* до ближайшего кратного параметра *Выражение2*.

Бинарные операторы выполняются слева направо и имеют меньший приоритет, чем унарные.

Команды в программе отладки adb

Команды **?** (вопросительный знак) и / (косая черта) позволяют просмотреть содержимое сегмента текста или данных. Команда **=** (знак равенства) показывает заданный адрес в указанном формате. Символ **?** и / могут применяться с последующим знаком ***** (звездочка).

?*Формат*
Показывает содержимое параметра *Объектный_файл*, начиная с параметра *Адрес*. Значение **.** (точки) увеличивается на сумму приращений всех символов формата.

/*Формат*
Показывает содержимое параметра *Файл_дампа*, начиная с параметра *Адрес*. Значение **.** (точки) увеличивается на сумму приращений всех символов формата.

=*Формат*
Показывает значение параметра *Адрес*. Буквы формата **i** и **s** игнорируются этой командой.

Параметр *Формат* состоит из одного или нескольких символов, определяющих стиль печати. Перед каждым символом формата можно указать десятичное целое число - его счетчик повторений. При пошаговой обработке формата **.** (точка) увеличивается на значение, заданное для каждого символа формата. Если формат не задан, то принимается последний применявшийся формат.

Допустимы следующие символы формата:

a Печатает значение **.** (точки) в символьной форме. При этом выполняется проверка типов символов.

b Печатает адресуемый байт как число без знака в текущей системе счисления.

c Печатает адресуемый символ.

C Печатает адресуемый символ в соответствии со следующими соглашениями об escape-символах:

- Символы печатаются в виде знака ~ (тильда), после которого следует соответствующий печатаемый символ.
- Непечатаемые символы печатаются в следующем виде: ~ (тильда) **<***число***>**, где *число* - шестнадцатеричное значение символа. Символ ~ печатается в виде ~ ~ (две тильды).

d Печатает числа в десятичном формате.

Д Печатает числа в длинном десятичном формате.

f Печатает 32-разрядные числа в формате с плавающей точкой.

F Печатает числа в формате двойной длины с плавающей точкой.

i *Число*
Печатает в виде инструкций. *Число* - это длина инструкции в байтах.

n Печатает новую строку.

o Печатает 2 байта в восьмеричном формате.

O Печатает 4 байта в восьмеричном формате.

p Печатает адресуемое значение в символьной форме в соответствии с теми же правилами символьного представления, что и для буквы формата **a**.

q Печатает 2 байта как число без знака в текущей системе счисления.

Q Печатает 4 байта как число без знака в текущей системе счисления.

r Печатает пробел.

s *Число*
Печатает адресуемый символ до обнаружения нуля.

S *Число*
Печатает строку в соответствии с соглашением о замене escape-символов знаком ~ (тильда). Переменная *Число* задает длину строки с учетом конечного нуля.

t При наличии предшествующего целого числа выполняет переход к следующей позиции табуляции, определяемой этим числом. Например, команда форматирования **8t** перемещает на следующую позицию табуляции через 8 пробелов.

u Печатает десятичное число без знака.

U Печатает длинное десятичное число без знака.

x Печатает 2 байта в шестнадцатеричном формате.

X Печатает 4 байта в шестнадцатеричном формате.

+ Печатает 4 байта в формате даты.

/ Локальный или глобальный символ данных.

? Локальный или глобальный символ текста.

= Локальный или глобальный абсолютный символ.

"..."
Печатает строку, указанную в кавычках.

^ Уменьшает **.** (точку) на текущее приращение. Печать не выполняется.

+ Увеличивает **.** (точку) на единицу. Печать не выполняется.

- Уменьшает **.** (точку) на единицу. Печать не выполняется.

Новая строка
Повторяет предыдущую команду с увеличением значения *Счетчик* на 1.

[?/]l*Значение Маска*
Применяет к словам, начинающимся с **.** (точки), маску, заданную параметром *Маска*, и выполняет сравнение с параметром *Значение* до тех пор, пока не будет найдено точное соответствие. Если указано **L**, то за один раз проверяется 4 байта, а не 2. Если найти соответствие не удается, то значение **.** (точка) останется прежним, в противном случае оно будет установлено равным найденному расположению. Если параметр *Маска* не указан, то по умолчанию принимается значение -1.

[?/]w*Значение***...**

Записывает 2-байтовый параметр *Значение* в адресуемое расположение. Если задана команда **W**, то будут записаны 4 байта, а если команда **V** - то 1 байт. При вызове команды **w** или **W** могут применяться ограничения, связанные с выравниванием.

[,*Счетчик***][?/]m B1 E1 F1[?/]**

Записывает новые значения для параметров *B1*,*E1* и *F1*. Если выражений меньше трех, то остальные параметры схемы изменены не будут. Если после **?** (вопросительного знака) или / (косой черты) указан символ ***** (звездочка), то будет изменен второй сегмент (*B2, E2, F2*) схемы. Если список завершается символом **?** или /, то для обработки последующих запросов будет применен файл *Объектный_файл* или *Файл_дампа*, соответственно. (Например, при выполнении команды **/m?** символ / указывает на файл *Объектный_файл*. Если указан параметр *Счетчик*, то команда **adb** изменяет схемы, связанные только с данным файлом или библиотекой. Команда **$m** показывает счетчик, соответствующий конкретному файлу. Если параметр *Счетчик* не указан, то будет принято значение по умолчанию - 0.

>*Имя*

Присваивает . (точку) переменной или регистру, заданным параметром *Имя*.

! Вызывает оболочку для чтения строки, следующей за символом **!** (восклицательный знак).

$ *модификатор*

Различные команды. Для *Модификатор* возможны следующие значения:

<*Файл* Считывает команды из указанного файла и возвращается к стандартному вводу. Если счетчик равен 0, то команда будет проигнорирована. До выполнения первой команды из указанного файла значение счетчика помещается в переменную **adb 9**.

<<*Файл*

Считывает команды из указанного файла и возвращается к стандартному вводу. Команду **<<***Файл* можно вызвать в файле, не закрывая его. Если счетчик равен 0, то команда будет проигнорирована. До выполнения первой команды из указанного файла значение счетчика помещается в переменную **adb 9**. Переменная **adb 9** сохраняется перед вызовом команды **<<***File* и восстанавливается после ее завершения. Число команд **<<***Файл*, вызываемых одновременно, ограничено.

>*Файл* Направляет вывод в указанный файл. Если параметр *Файл* опущен, то будет создан стандартный вывод. Если параметр *Файл* не существует, то он будет создан.

b Печатает все точки прерывания и связанные с ними счетчики и команды.

c Печатает список кадров стека. Если задан параметр *Адрес*, то он принимается в качестве адреса текущего кадра (вместо регистра указателя кадра). Если указана буква формата **C**, то будут напечатаны имена и значения всех автоматических и статических переменных для всех активных функций. Если указан параметр *Счетчик*, то будет напечатано только указанное в нем число кадров.

d Устанавливает текущее основание системы счисления равным значению *Адрес*, либо 16, если адрес не указан.

e Печатает имена и значения внешних переменных. Если задан счетчик, то будут напечатаны только те внешние переменные, которые связаны с печатаемым файлом.

f Печатает регистры с плавающей точкой в шестнадцатеричном формате.

i *набор-инструкций*

Выбирает набор инструкций для дезассемблирования.

I Изменяет каталог по умолчанию, задаваемый флагом **-I**, на значение параметра *Имя*.

m Печатает схему адресов.

n *набор-мнемоник*

Выбирает мнемоники для дезассемблирования.

o Устанавливает основание системы счисления равным 8.

q Завершает выполнение команды **adb**.

r Печатает основные регистры и инструкции, адресуемые **iar**, а также устанавливает **.** (точку) равной **iar**. Параметр *Номер***$r** печатает регистр, задаваемый переменной *Номер*. Параметр *Номер*,*Счетчик***$r** печатает регистры *Номер*+*Счетчик*-1,...,*Номер*.

s Устанавливает ограничение на число символьных совпадений для значения *Адрес*. Значение по умолчанию - 255.

v Печатает все ненулевые переменные в восьмеричном формате.

w Задает ширину страницы вывода для параметра *Адрес*. Значение по умолчанию - 80.

P *Имя* Вводит значение *Имя* в строку приглашения.

? Печатает ИД процесса, сигнал, вызвавший остановку или завершение, и регистры **$r**.

: *Модификатор*

Управляет подпроцессом. Допустимы следующие модификаторы:

b*Команда*

Задает точку прерывания в параметре *Адрес*. Всего точка прерывания запускается *Счетчик*-1 раз. При каждом прохождении точки прерывания вызывается указанная команда. Если эта команда установит **.** (точку) рядом со значением 0, то выполнение программы в этой точке будет остановлено.

c*Сигнал*

Продолжает подпроцесс указанным сигналом. Подпроцесс будет продолжен с адреса, заданного параметром *Адрес* (если он указан). Если сигнал не указан, то будет передан сигнал, вызвавший остановку процесса. Пропуск точек прерывания выполняется так же, как и в случае модификатора **r**.

d Удаляет точку прерывания в параметре *Адрес*.

k Останавливает текущий подпроцесс, если он выполняется.

r Запускает программу, указанную в параметре *Объектный_файл*, в качестве подпроцесса. Если параметр *Адрес* задан явно, то программа будет запущена с этой точки. В противном случае программа будет запущена со стандартной точки входа. Параметр *Счетчик* задает количество точек входа, которое будет проигнорировано до остановки подпроцесса. Аргументы для подпроцесса можно задать в одной строке с командой. Аргумент, начинающийся с символа **<** или **>**, задает стандартный ввод или вывод. В начале выполнения подпроцесса все сигналы включены.

s*сигнал*

Продолжает пошаговое выполнение подпроцесса. Число шагов задается параметром *Счетчик*. Если в настоящее время никакие подпроцессы не выполняются, то в качестве подпроцесса будет запущен указанный *Объектный_файл*. В этом случае сигналы передаваться не будут. Остаток строки передается подпроцессу в качестве аргументов.

Переменные в программе отладки adb

В команде **adb** можно задать некоторые переменные. При запуске программы **adb** значения для перечисленных ниже переменных берутся из системного заголовка в указанном файле дампа. Если файл дампа (**core**) задается не параметром *файл-дампа*, то эти значения берутся из параметра *объектный-файл*:

0 Последнее напечатанное значение

1 Последний блок смещения в источнике инструкций

2 Предыдущее значение переменной **1**

9 Счетчик последней команды **$<** или **$<<**

b Базовый адрес сегмента данных

d Размер сегмента данных

e Адрес входа программы

m Сигнатура

s Размер сегмента стека

t Размер текстового сегмента

Пример программы adb: adbsamp

Ниже приведен пример программы:

```
/* Распечатка программы adbsamp.c */
char  str1[ ] = "Это символьная строка";
int one = 1;
int number = 456;
long lnum = 1234;
float fpt = 1.25;
char str2[ ] = "Это вторая символьная строка";
main()
{
        one = 2;
        printf("Первая строка = %s\n",str1);
        printf("Единица = %d\n",one);
        printf("Число = %d\n",lnum);
        printf("Число с плавающей точкой = %g\n",fpt);
        printf("Вторая строка = %s\n",str2);
}
```

Откомпилируйте программу с помощью команды **cc** и запишите ее в файл **adbsamp** :

```
cc -g adbsamp.c -o adbsamp
```

Для запуска сеанса отладки введите следующую команду:

```
adb adbsamp
```

Пример программы adb: adbsamp2

Ниже приведен пример программы:

```
/* Распечатка программы adbsamp2.c*/
int     fcnt,loop_count;

f(a,b)
int a,b;
{
        a = a+b;
        fcnt++;
        return(a);
}
main()
{
        loop_count = 0;
        while(loop_count <= 100)
        {
                loop_count = f(loop_count,1);
                printf("%s%d\n","Счетчик цикла: ", loop_count);
                printf("%s%d\n","Счетчик цикла fcnt: ",fcnt);
        }
}
```

Откомпилируйте программу с помощью команды **cc** и сохраните ее в файле **adbsamp2**:

```
cc -g adbsamp2.c -o adbsamp2
```

Для запуска сеанса отладки введите следующую команду:

```
adb adbsamp2
```

Пример программы adb: adbsamp3

В приведенный ниже пример программы **adbsamp3.c** включен бесконечный рекурсивный вызов функций.

При попытке выполнить эту программу до конца возникнет страничная ошибка, которая приведет к аварийному завершению программы.

```
int     fcnt,gcnt,hcnt;
h(x,y)
int x,y;
{
        int hi;
        register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++;
        hj:
        f(hr,hi);
}
g(p,q)
int p,q;
{
        int gi;
        register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++;
        gj:
        h(gr,gi);
}
f(a,b)
int a,b;
{
        int fi;
        register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++;
        fj:
        g(fr,fi);
}
main()
{
        f(1,1);
}
```

Откомпилируйте программу с помощью **cc** и запишите ее в файл **adbsamp3**:

```
cc -g adbsamp3.c -o adbsamp3
```

Для запуска сеанса отладки введите следующую команду:

```
adb adbsamp3
```

Пример дампа i-узла и каталога при отладке с помощью adb

В данном примере показано, как создавать сценарии **adb** для просмотра содержимого каталогов и таблицы i-узлов файловой системы. В примере используется каталог с именем **dir**, который содержит разнообразные файлы.

Файловая система связана с файлом устройства /dev/hd3 (**/tmp**), причем предполагается, что у пользователя есть права на чтение этого файла.

Для просмотра каталога нужно создать сценарий. Обычно каталог содержит одну или несколько записей. Каждая запись состоит из беззнакового номера i-узла файла и 14-символьного имени файла. Для просмотра этой информации в файле сценария нужно указать соответствующую команду. Программа **adb** предназначена для работы с объектными файлами в формате **xcoff**. Этот формат отличен от формата каталога. Из-за этого программа **adb** решит, что длина его текстового сегмента равна нулю. С помощью команды **m** укажите программе **adb**, что длина текстового сегмента каталога больше нуля. Начните сеанс работы с **adb** с ввода команды:

```
,0?m 360 0
```

Например, для вывода первых 20 записей, в которых номер i-узла файла будет отделяться от его имени знаком табуляции, нужно ввести следующую команду:

```
0,20?ut14cn
```

Вместо второго числа (20) нужно указать реальное число записей. Если в начале сценария будет указана приведенная ниже команда, то программа **adb** выведет указанные строки как заголовки столбцов значений:

```
="inumber"8t"Name"
```

После создания файла сценария укажите его в качестве файла ввода при запуске **adb** с именем вашего каталога. Например, для того чтобы программа **adb** была выполнена для каталога geo, считывая команды из входного файла сценария ddump, нужно ввести следующую команду:

```
adb geo - <ddump
```

Знак минус (-) запрещает программе **adb** открывать файл дампа. Программа **adb** считывает команды из файла сценария.

Для просмотра таблицы i-узлов файловой системы создайте новый сценарий и запустите программу **adb**, указав в качестве параметра файл устройства файловой системы. Таблица i-узлов файловой системы имеет сложную структуру. Каждая запись содержит:

- Флаги состояния (одно слово)
- Число связей (один байт)
- ИД пользователя и группы (по 2 байта)
- Размер (байт и слово)
- Адреса блоков файла на диске (8 слов)
- Дата создания и изменения (2 слова)

Ниже приведен пример вывода дампа каталога:

```
        inumber имя
0:       26    .
        2..
        27    .estate
        28    adbsamp
        29    adbsamp.c
        30    calc.lex
        31    calc.yacc
        32    cbtest
        68    .profile
        66    .profile.bak
        46    adbsamp2.c
        52    adbsamp2
        35    adbsamp.s
        34    adbsamp2.s
        48    forktst1.c
        49    forktst2.c
```

```
50    forktst3.c
51    lpp&us1.name
33    adbsamp3.c
241   sample
198   adbsamp3
55    msgqtst.c
56    newsig.c
```

Таблица i-узлов начинается с адреса 02000. Вы можете просмотреть таблицу, начиная с первой записи, указав в файле сценария следующую команду:

```
02000,-1?on3bnbrdn8un2Y2na
```

Для того чтобы данные вывода было легче читать, в команде указано несколько символов новой строки.

Для просмотра таблицы i-узлов файла **/dev/hd3** с помощью файла script введите следующую команду:

```
adb /dev/hd3 - <script
```

На экране появится набор записей следующего вида:

```
02000: 073145
    0163 0164 0141
    0162 10356
    28770 8236 25956 27766 25455 8236 25956 25206
    1976 Feb 5 08:34:56 1975 Dec 28 10:55:15
```

Пример форматирования данных при отладке с помощью adb

Для того чтобы после каждой машинной инструкции выдавался текущий адрес, введите следующую команду:

```
main , 5 ? ia
```

Если эта команда используется с приведенным здесь примером программы **adbsamp**, то ее вывод будет выглядеть следующим образом:

```
.main:
.main:          mflr 0
.main+4:        st r0,  0x8(r1)
.main+8:        stu rs,    (r1)
.main+c:        li l  r4,  0x1
.main+10:       oril  r3, r4, 0x0
.main+14:
```

Для того чтобы показать, что текущий адрес не относится к команде, стоящей с ним в одной строке, добавьте к вводимой команде символ перехода на новую строку (n):

```
.main , 5 ? ian
```

Перед символом форматирования можно указать число повторений данного формата.

Для просмотра списка инструкций с указанием адреса каждой четвертой инструкции введите следующую команду:

```
.main,3?4ian
```

Если эта команда будет выполнена для примера программы **adbsamp**, то будут показаны следующие данные:

```
.main:
                mflr 0
                st  r0, 0x8(r1)
                stu r1, -56(r1)
                lil r4, 0x1

.main+10:
```

```
                oril r3, r4, 0x0
                bl .f
                l r0, Ox40(r1)
                ai r1, r1, 0x38

.main+20:
                mtlr r0
                br
                Invalid opcode
                Invalid opcode

.main+30:
```

Внимательно выбирайте позицию числа в аргументах команды.

Следующая команда похожа на предыдущую, однако ее вывод будет другим:

```
main,3?i4an
```

```
.main:
.main:          mflr  0
.main+4:        .main+4:        .main+4:        .main+4:
                st  r0, 0x8(r1)
.main+8:        .main+8:        .main+8:        .main+8:
                stu  r1,    (r1)
.main+c:        .main+c:        .main+c:        .main+c:
```

Можно комбинировать форматы и выдавать на экран информацию в достаточно сложном виде. Например, для просмотра мнемонических имен команд и их шестнадцатеричных эквивалентов можно ввести такую команду:

```
.main,-1?i^xn
```

В этом примере выдача начинается с адреса `main`. Отрицательное число (-1) приводит к вызову команды в бесконечном цикле, так что вывод на дисплей продолжается до тех пор, пока не произойдет ошибка (например, будет достигнут конец файла). В записи формата `i` выдает мнемоническую команду, ^ (знак вставки) перемещает текущий адрес к началу команды, а `x` выдает шестнадцатеричный эквивалент команды. Наконец, `n` отправляет на терминал символ новой строки. Вывод имеет примерно следующий вид:

```
.main:
.main:          mflr  0
                7c0802a6
                  st  r0, 0x8(r1)
                9001008
                  st r1, -56(r1)
                9421ffc8
                  lil r4, 0x1
                38800001
                  oril r3, r4, 0x0
                60830000
                  bl  -  .f
                4bffff71
                  l r0, 0x40(r1)
                80010040
                  ai r1, r1, 0x38
                30210038
                  mtlr r0
                7c0803a6
```

В следующем примере показано, как можно комбинировать форматы в командах **?** или **/** для выдачи на экран различных типов значений, хранящихся в одной и той же программе. В примере используется программа **adbsamp**. Для выбора переменных необходимо сначала задать точку прерывания для останова программы, а затем запустить программу, которая будет выполнена до точки прерывания. Точка прерывания устанавливается с помощью команды **:b**:

```
.main+4:b
```

Для того чтобы убедиться, что данная точка прерывания установлена, воспользуйтесь командой **$b**:

```
$b
точки прерывания
count  bkpt      command
1    .main+4
```

Запустите программу и дождитесь останова в точке прерывания:

```
:r
adbsamp: running
breakpoint   .main+4:    st r0, 0x8(r1)
```

Теперь вы можете просматривать информацию о состоянии программы. Для выдачи на экран значения какой-либо переменной укажите в команде / (косая черта) имя переменной и требуемый формат. Например, для просмотра содержимого переменной `str1` в виде строки нужно ввести следующую команду:

```
str1/s
str1:
str1:     Это строка символов
```

Для просмотра переменной `number` в виде десятичного целого числа нужно ввести следующую команду:

```
number/D
number:
number:    456
```

Можно просматривать значения переменных в различных форматах. Например, с помощью следующих команд вы можете вывести на экран значение переменной **lnum** как 4-байтовое десятичное, восьмеричное или шестнадцатеричное число:

```
lnum/D
lnum:
lnum:     1234

lnum/O
lnum:
lnum:     2322

lnum/X
lnum:
lnum:     4d2
```

Можно просматривать значения переменных и в других форматах. В следующем примере выдается последовательность шестнадцатеричных значений некоторых переменных, занимающая пять строк, по восемь значений в строке:

```
str1,5/8x
str1:
str1:   5468  6973  2069  7320  6120  6368  6172  6163
         7465  7220  7374  7269  6e67  0   0   0     0

number:  0      1c8     0     0     0    d2     0     0
         3fa0     0     0     0  5468  6973  2069  7320
         7468  6520  7365  636f  6e64  2063  6861  7261
```

Поскольку данные могут содержать комбинацию числовых и строковых значений, то можно выводить каждое значение и как число, и как символ, чтобы можно было видеть, где в действительности находятся строки. Это можно сделать с помощью команды

```
str1,5/4x4^8Cn
str1:
str1:    5468  6973  2069  7320      Это стр
          6120  6368  6172  6163       ока симв
          7465  7220  7374  7269       олов~@~@
          6e67     0     0     1       ~@~@~@~@~@~@~@~A
          0 1c8 0 0 ~@~@~A~<c8>~@~@~@~@
```

В данном случае команда выдает четыре значения в шестнадцатеричном виде, а затем повторяет те же значения как восемь символов ASCII. Символ ^ (знак вставки) используется четыре раза перед выдачей символов, чтобы снова сделать текущим начальный адрес данной строки.

Для того чтобы можно было легко разобраться в данных, показанных на дисплее, вставляйте между числами и символами символы табуляции, и печатайте адрес каждой строки:

```
str1,5/4x4^8t8Cna
str1:
str1:     5468  6973  2069  7320      Это стр
str1+8:   6120  6368  6172  6163      ока симв
str1+10:  7465  7220  7374  7269      олов~@~@
str1+18:  6e67     0     0     1      ~@~@~@~@~@~@~A

number:
number: 0 1c8 0 0 ~@~@~A~<c8>~@~@~@~@
fpt:
```

Пример трассировки нескольких функций при отладке посредством adb

Ниже приведен пример выполнения программы под управлением **adb**, демонстрирующий основные операции отладки. Описание этих операций приведено в следующих разделах.

Примечание: Пример программы **adbsamp3**, рассматриваемый в данном разделе, содержит бесконечную рекурсию. При попытке выполнить эту программу до конца возникнет страничная ошибка, которая приведет к аварийному завершению программы.

Исходная программа для данного примера хранится в файле с именем **adbsamp3.c**. С помощью команды **cc** откомпилируйте этот файл и создайте исполняемый файл **adbsamp3**:

```
cc adbsamp3.c -o adbsamp3
```

Запуск программы adb

Для запуска сеанса отладки и открытия файла с программой введите следующую команду (файл дампа не используется):

```
adb adbsamp3
```

Установка точек прерывания

Установите точки прерывания в начале каждой функции с помощью команды **:b**:

```
.f:b
.g:b
.h:b
```

Просмотр набора команд

Просмотрите первые пять команд функции **f**:

```
.f,5?ia
.f:
.f:             mflr  r0
.f+4:           st  r0, 0x8(r1)
.f+8:           stu  r1, -64(r1)
.f+c:           st  r3,  0x58(r1)
.f+10:          st  r4,  0x5c(r1)
.f+14:
```

Просмотрите первые пять команд функции **g** без указания адресов:

```
.g,5?i
.g:    mflr  r0
        st  r0, 0x8(r1)
        stu  r1, -64(r1)
        st  r3,  0x58(r1)
        st  r4,  0x5c(r1)
```

Запуск программы adsamp3

Для запуска программы введите команду

```
:r
adbsamp3: running
breakpoint   .f:       mflr  r0
```

Программа **adb** будет выполнять пример программы до первой точки прерывания. В этот момент она остановится.

Удаление точки прерывания

Поскольку при выполнении программы до этой точки прерывания ошибок не произошло, первую точку прерывания можно удалить:

```
.f:d
```

Продолжение выполнения программы

Для возобновления работы программы введите команду **:c**:

```
:c
adbsamp3: running
breakpoint   .g:       mflr  r0
```

Программа **adb** продолжит выполнение **adbsamp3** со следующей команды. Она снова остановится, когда дойдет до следующей точки прерывания.

Трассировка последовательности выполнения

Для трассировки последовательности выполнения введите команду

Просмотр значения переменной

Для просмотра значения целочисленной переменной `fcnt` введите следующую команду:

```
fcnt/D
fcnt:
fcnt:     1
```

Пропуск точек прерывания

Затем продолжите выполнение программы и пропустите первые 10 точек прерывания:

```
,10:c
adbsamp3: running
breakpoint   .g:       mflr  r0
```

Программа **adb** запустит программу **adbsamp3** и снова выдаст сообщение о выполнении. Выполнение не прекратится до тех пор, пока не будут пройдены ровно 10 точек прерывания. Для того чтобы убедиться, что эти точки прерывания были пропущены, просмотрите информацию о трассировке:

```
$c
.g(0,0) .f+2a
.f(2,11) .h+28
.h(10,f) .g+2a
```

```
.g(11,20) .f+2a
.f(2,f) .h+28
.h(e,d) .g+2a
.g(f,1c) .f+2a
.f(2,d) .h+28
.h(c,b) .g+2a
.g(d,18) .f+2a
.f(2,b) .h+28
.h(a,9) .g+2a
.g(b,14) .f+2a
.f(2,9) .h+28
.h(8,7) .g+2a
.g(9,10) .f+2a
.f(2,7) .h+28
.h(6,5) .g+2a
.g(7,c) .f+2ae
.f(2,5) .h+28
.h(4,3) .g+2a
.g(5,8) .f+2a
.f(2,3) .h+28
.h(2,1) .g+2a
.g(2,3) .f+2a
.f(1,1) .main+e
.main(0,0,0) start+fa
```

Обзор программы символьной отладки dbx

Программа символьной отладки **dbx** позволяет отлаживать прикладные программы на двух уровнях: на уровне исходного кода и на уровне ассемблера. В режиме отладки на уровне исходного кода можно отлаживать программы, написанные на языках C, C++ и FORTRAN.

В режиме отладки на уровне ассемблера можно отлаживать исполняемые программы на машинном уровне. Команды, применяемые для обоих типов отладки, схожи.

С помощью программы отладки **dbx** можно осуществлять пошаговое выполнение прикладной программы или расставить точки прерывания в объектной программе, на которых будет останавливаться программа отладки. Можно также выполнять поиск и просматривать фрагменты исходных файлов прикладной программы.

Информация о способах выполнения различных задач с помощью программы отладки **dbx** приведена в следующих разделах:

Работа с программой отладки dbx

В этом разделе приведена информация о работе с программой отладки **dbx**.

Запуск программы отладки dbx

Программу **dbx** можно запустить с различными параметрами. Ниже перечислены три самых распространенных способа запуска **dbx**:

- Вызов команды **dbx** для конкретного объектного файла
- Вызов команды **dbx** с флагом **-r** для отладки программы, завершившейся аварийно
- Вызов команды **dbx** с флагом **-a** для отладки процесса, который уже запущен

Сразу после запуска команда **dbx** ищет файл **.dbxinit** в текущем пользовательском каталоге или в пользовательском каталоге **$HOME**. Если файл **.dbxinit** существует, то в начале сеанса отладки выполняются команды из этого файла. Если файл **.dbxinit** есть и в домашнем, и в текущем каталоге, то оба файла считываются в указанном порядке. Так как файл **.dbxinit** из текущего каталога считывается вторым, то команды из этого файла могут изменить результат выполнения команд из файла, расположенного в домашнем каталоге.

Если имя объектного файла не указано, программа **dbx** запрашивает имя объектного файла для отладки. Имя файла по умолчанию - **a.out**. Если в текущем каталоге есть файл **core** или если указан параметр *ФайлДампа*, то программа **dbx** сообщает о том, в каком месте произошел сбой программы. До начала выполнения объектного файла можно просмотреть значения переменных, а также содержимое регистров и областей памяти, сохраненные в файле дампа. Начиная с этого момента, **dbx** начинает запрашивать команды.

Отладка образа ядра с недостающими зависимыми модулями

Начиная с AIX 5.3, программа **dbx** позволяет анализировать образ ядра даже в отсутствие зависимых модулей. При инициализации для каждого из недостающих модулей выдается сообщение.

В обычном режиме программа **dbx** использует информацию из текстовых разделов и таблиц символов зависимых модулей. Поскольку эта информация в таких ситуациях будет доступна не полностью, на сеансы **dbx** в отсутствие зависимых модулей накладываются следующие ограничения:

- При попытке прочитать содержимое областей памяти, относящихся к текстовым разделам недостающих модулей, выдается сообщение об ошибке. Это сообщение схоже с сообщением об ошибке, вызванной отсутствием запрошенных данных в файле ядра.
- Пользователю недоступна информация о символах, хранящихся в таблицах символов недостающих модулей. Поведение программы **dbx** в таких ситуациях такое же, как если бы из недостающего модуля была исключена таблица символов.
- Фреймы стеков процедур из недостающих модулей выглядят в программе следующим образом:

  ```
  .()
  ```

  Кроме того, указываются адрес инструкции в неизвестной процедуре и имя недостающего модуля.

Программе **dbx** можно указать расположение доступных из недостающих модулей с помощью флага **-p**. Дополнительная информация приведена в описании команды **dbx** в книге *Справочник по командам, том 2*.

Отладка образа ядра с рассинхронизированными зависимыми модулями

Начиная с AIX 5.3, программа **dbx** определяет, изменялись ли зависимые модули, указанные в файле ядра, с момента создания файла ядра. При инициализации программы для каждого измененного зависимого модуля выдается уведомление.

Следует помнить, что вся информация, которую программа **dbx** показывает на основе содержимого измененного зависимого модуля, может быть неточной. Для того чтобы поставить пользователя в известность о возможных неточностях, программа **dbx** выдает уведомления всегда, когда отображается потенциально неточная информация.

Можно отключить эту функцию и перевести программу **dbx** в режим, когда измененные зависимые модули считаются отсутствующими. Для этого нужно экспортировать переменную среды *DBX_MISMATCH_MODULE* со значением DISCARD. В этом режиме программа **dbx** по-прежнему уведомляет пользователя о том, что модуль был изменен, но в дальнейшем работает так, как если бы он был недоступен.

Программе **dbx** можно указать расположение правильных версий недостающих модулей с помощью флага **-p**. Дополнительная информация приведена в описании команды **dbx** в книге *Справочник по командам, том 2*.

Запуск команд оболочки из dbx

С помощью подкоманды **sh** можно запускать команды оболочки, не прерывая программу отладки.

Если команда в **sh** не указана, то будет выполнен временный выход в оболочку, а после завершения работы с ней управление будет возвращено программе **dbx**.

Редактор командной строки dbx

В командной строке **dbx** применяются функции редактирования, аналогичные функциям оболочки **Korn**. Режим **vi** предоставляет функции, схожие с редактором **vi**, а режим **emacs** - схожие с редактором **emacs**.

Для включения этих функций вызовите подкоманду **dbx set -o** или **set edit**. Для включения функций редактора **vi** введите подкоманду `set edit vi` или `set -o vi`.

Кроме того, выбрать режим редактирования можно с помощью переменной среды **EDITOR**.

Программа **dbx** записывает введенные команды в файл хронологии **.dbxhistory**. Если переменная среды **DBXHISTFILE** не установлена, то будет применяться файл хронологии **$HOME/.dbxhistory**.

По умолчанию команда **dbx** хранит 128 последних введенных команд. Для увеличения этого значения применяется переменная среды **DBXHISTSIZE**.

Управление выполнением программ

При работе с программой символьной отладки **dbx** вы можете добавлять в программу точки прерывания. После запуска программы **dbx** вы можете задать строки и адреса, которые будут выполнять роль точек прерывания, а затем запустить программу, отлаживаемую с помощью **dbx**. Когда программа дойдет до точки прерывания, ее выполнение будет приостановлено с выдачей соответствующего сообщения. После этого можно будет проверить состояние программы с помощью команд **dbx**.

Вместо применения точек прерывания можно выполнять программу по шагам, т.е. по одной инструкции или по одной строке за раз.

Установка и удаление точек прерывания

С помощью подкоманды **stop** задайте точки прерывания в программе **dbx**. Команда **stop** останавливает выполнение прикладной программы при выполнении определенных условий:

- *Переменная* изменилась и задан параметр *Переменная*.
- Выполнено *Условие* и задан флаг **if** *Условие*.
- Вызвана *Процедура* и задан флаг **in** *Процедура*.
- Достигнута *строка кода* и задан флаг **at***строка кода*.

> **Примечание:** Значение *строки кода* может быть указано в виде целого числа или в виде строки, содержащей имя исходного файла, затем двоеточие (:) и целое число, задающее номер строки в файле.

После вызова этих команд программа **dbx** будет отправлять сообщение с номером одного из событий, заданных в качестве точки прерывания. Можно связать команды dbx с идентификатором определенного события с помощью команды **addcmd**. Эти связанные команды dbx выполняются при достижении точки прерывания, точки трассировки или точки наблюдения, соответствующей данному событию. С помощью команды **delcmd** можно удалить связанные команды **dbx** из указанного ИД события.

Запуск программ

Подкоманда **run** предназначена для запуска программы. Эта команда указывает программе **dbx**, что необходимо запустить объектный файл. Аргументы в этом случае указываются так же, как и при запуске из командной строки. Формат подкоманды **rerun** совпадает с форматом **run**, за тем исключением, что если ей

не передан список аргументов, то используются аргументы из предыдущего вызова. После запуска отлаживаемая программа выполняется до наступления любого из следующих событий:

- Программа встречает точку прерывания
- Возникает сигнал, который не должен игнорироваться, например, **INTERRUPT** или **QUIT**.
- При отладке нескольких процессов возникло событие, связанное с несколькими событиями.
- Программа выполняет функцию **load**, **unload** или **loadbind**.

  Примечание: Программа **dbx** игнорирует это условие, если задана переменная отладки **$ignoreload**. По умолчанию эта переменная задана. Более подробная информация приведена в описании команды **set**.
- Программа завершает работу

В любом случае управление передается программе отладки **dbx**, которая показывает сообщение о причине остановки программы.

Продолжить выполнение остановленной программы можно с помощью следующих команд:

Команда	Описание
cont	Продолжает выполнение программы с той точки, в которой оно было прервано.
Отключить	Продолжает выполнение программы с той точки, в которой оно было прервано, при этом происходит выход из отладчика. Эта команда полезна в тех случаях, если вы исправили программу и хотите продолжить ее выполнение, но уже не в среде отладчика.
return	Продолжает выполнение до тех пор, пока не произойдет возврат в *Процедуру*, или, если *Процедура* не указана, пока не завершится текущий процесс.
skip	Продолжает выполнение программы до ее завершения или до перехода через точки прерывания указанное *число* + 1 раз.
step	Выполняет одну строку исходного кода или указанное *число* строк.
next	Выполняет программу до следующей строки исходного кода или выполняет указанное *число* строк исходного кода.

Наиболее распространенный способ отладки - это пошаговое выполнение программы. Для этого предназначены подкоманды **step** и **next**. Различие между этими двумя командами проявляется лишь в том случае, когда следующая строка исходного кода содержит вызов функции. В этом случае команда **step** входит в функцию и останавливается, а как команда **next** полностью выполняет функцию и останавливается на следующем операторе после вызова функции.

Изменить способ выполнения программы **step** можно с помощью переменной **$stepignore**. Дополнительная информация приведена в документации по команде **dbx** в *Справочник по командам, том 2*.

Остановкам в ходе пошагового выполнения программы не присваивается номер события, так как останов программы не связан ни с каким постоянным событием.

Если в программе есть несколько нитей, подкоманды **cont**, **next**, **nexti** и **step** обработают их правильно. Эти команды выполняются для конкретной нити, поэтому даже если другая нить обрабатывается командами **cont**, **next**, **nexti** или **step**, первая нить будет продолжать работать до тех пор, пока не встретит код, вызвавший прерывание.

Если вам нужно, чтобы перечисленные команды выполняли только текущую нить, можно установить переменную отладчика **dbx** с именем **$hold_next**; если эта переменная установлена, программа **dbx** во время выполнения команд **cont**, **next**, **nexti** и **step** приостанавливает обработку других нитей.

Примечание: При работе в таком режиме следует помнить, что приостановленная нить не может снять установленные ею блокировки; если при этом другая нить ожидает снятия одной из этих блокировок, выполнение программы может зайти в тупик.

Отделение вывода dbx от вывода программы

При отладке программ, работающих с экраном, например, текстовых редакторов и графических программ, следует применять команду **screen**. Эта команда открывает для программы **dbx** окно Xwindow.

Отлаживаемая программа при этом продолжает выполняться в своем окне. Если не вызвана команда **screen**, то вывод программы **dbx** будет перемешан с выводом других программ.

Трассировка

Подкоманда **trace** указывает программе **dbx**, что необходимо напечатать информацию о состоянии отлаживаемой программы. Подкоманда **trace** может существенно замедлить выполнение программы, в зависимости от загруженности **dbx**. Существует пять вариантов трассировки:

- Пошаговое выполнение программы, позволяющее печатать каждую выполняемую строку исходного кода. Для настройки команды **trace** применяется переменная отладки **$stepignore**. Более подробная информация приведена в описании команды **set**.
- Отладчик может печатать только выполняемые строки конкретной процедуры. Можно также указать дополнительные условия, позволяющие управлять выводом трассировочной информации.
- Отладчик может отправлять сообщение при каждом вызове и завершении процедуры.
- Отладчик может напечатать указанную строку исходного кода, когда она будет выполняться.
- Отладчик может напечатать значение выражения, когда программа будет выполнять указанную строку.

Удаление событий трассировки происходит так же, как и удаление точек прерывания. При выполнении команды **trace** на экран выводится ИД соответствующего события, а также информация о внутреннем представлении этого события.

Просмотр и редактирование исходного файла с помощью программы отладки dbx

В этом разделе рассмотрен процесс работы с исходными файлами в программе отладки dbx.

С помощью программы **dbx** можно выполнять поиск и просматривать на экране фрагменты исходных файлов программы.

Для выполнения поиска необязательно открывать текст исходного файла. Программа **dbx** отслеживает текущий файл, текущую процедуру и текущую строку. Если существует файл дампа, то начальной текущей строкой текущего файла будет строка файла, содержащая исходную команду, на которой остановился процесс. это справедливо только в том случае, если процесс был прерван в разделе программы, откомпилированном для отладки.

Изменение каталога исходных файлов

По умолчанию программа **dbx** ищет исходный файл отлаживаемой программы в следующих каталогах:

- Каталог, в котором находился исходный файл во время компиляции. Поиск в этом каталоге выполняется только в том случае, если компилятор поместил в объект имя каталога исходных файлов.
- Текущий рабочий каталог.
- Каталог, в котором в данный момент находится программа.

Список просматриваемых каталогов можно изменить с помощью опции **-I**, указываемой при вызове **dbx**, или с помощью команды **use** в программе **dbx**. Например, если после компиляции исходного файла вы переместили его в другой каталог, то с помощью одной из этих команд вы должны указать его старое расположение, новое расположение и некоторое временное расположение.

Просмотр текущего файла

Просмотреть текст исходного файла можно с помощью команды **list**.

С командами **list**, **stop** и **trace** можно использовать символы $ (знак доллара) и @ (коммерческое 'at'), которые определяют выражение *строка-исходного-файла*. Символ $ задает следующую выполняемую, а символ @ - следующую показываемую строку.

Команда **move** предназначена для перехода к следующей указанной строке.

Изменение текущего файла или процедуры

Команды **func** и **file** предназначены для изменения текущего файла, текущей процедуры и текущей строки внутри программы **dbx** без запуска какой-либо части вашей программы.

В текущем файле можно выполнять поиск текста, совпадающего с регулярными выражениями. Если обнаружено совпадение, текущая строка заменяется строкой, содержащей найденный текст. Синтаксис команды поиска следующий:

/*Выражение* **[/]**
: Выполняется поиск по образцу в текущем исходном файле в прямом направлении.

? *RegularExpression* **[?]**
: Выполняется поиск по образцу в текущем исходном файле в обратном направлении.

Если аргументы не указаны, то команда **dbx** выполняет поиск регулярного выражения, которое было задано последним. При достижении конца (начала) файла происходит переход к его началу (концу).

С помощью команды **edit** можно загрузить исходный файл во внешний текстовый редактор. Можно переопределить редактор, установленный по умолчанию (**vi**), указав имя редактора в переменной среды **EDITOR** перед запуском программы **dbx**.

После завершения сеанса редактирования управление процессом будет вновь передано программе **dbx**.

Отладка программ с несколькими нитями

Программы с несколькими пользовательскими нитями вызывают функцию **pthread_create**. Когда процесс вызывает эту функцию, операционная система создает внутри процесса новую нить. При отладке программы с несколькими нитями необходимо работать не с процессами, а с отдельными нитями. Программа **dbx** работает только с пользовательскими нитями: в документации по **dbx** под словом *нить* обычно подразумевается *пользовательская нить*. Программа **dbx** присваивает каждой нити в отлаживаемом процессе уникальный номер. Кроме того, при работе с **dbx** следует различать понятия активной нити и текущей нити:

Активная нить
: Пользовательская нить, которая ответственна за останов программы при попадании в точку прерывания. С активными нитями работают команды, предназначенные для пошагового выполнения программы.

Текущая нить
: Пользовательская нить, которую вы изучаете. В контексте текущей нити работают команды, выдающие информацию.

По умолчанию активная нить и текущая нить - это одно и то же. С помощью команды **thread** можно сделать текущей другую нить. В списке нитей, выдаваемом командой **thread**, текущая нить отмечена знаком **>**. Если активная нить не совпадает с текущей, то она отмечена знаком *****.

Отладка программ с несколькими процессами

Программы с несколькими процессами обращаются к функциям **fork** и **exec**. Когда программа порождает процесс, операционная система создает новый процесс с тем же образом, что и у исходного процесса. Исходный процесс называется родительским, а создаваемый процесс - дочерним.

При выполнении процессом функции **exec** управление от исходного процесса передается новой программе. В обычных условиях выполняется отладка только родительского процесса. Однако программа **dbx** может выполнять отладку и дочерних процессов, если ввести команду **multproc**. Команда **multproc** позволяет выполнять параллельную отладку нескольких процессов.

Если в режиме параллельной отладки порождается новый процесс, то родительский и дочерний процессы останавливаются. Для контроля за работой дочернего процесса для программы **dbx** открывается отдельный виртуальный терминал Xwindow:

```
(dbx) multproc on
(dbx) multproc
включен режим параллельной отладки
(dbx) run
```

При порождении процесса выполнение родительского процесса останавливается и программа **dbx** показывает информацию о состоянии программы:

```
порожден новый процесс, pid = 422, процесс остановлен, ожидается ввод
остановлен из-за порождения дочернего процесса в режиме с несколькими
процессами в точке 0x1000025a (fork+0xe)
(dbx)
```

Затем для отладки дочернего процесса открывается еще один виртуальный терминал Xwindow:

```
отладка дочернего процесса, pid=422, процесс остановлен, ожидается ввод
остановлен из-за порождения дочернего процесса в режиме с несколькими
процессами в точке 0x10000250
10000250 (fork+0x4) )80010010    1       r0,0x10(r1)
(dbx)
```

В этой точке работают два различных сеанса отладки. В сеансе отладки дочернего процесса сохраняются все точки прерывания, установленные для родительского процесса, но повторно запускаться может только родительский процесс.

Если программа выполняет функцию **exec** в режиме параллельной отладки, то она переписывает саму себя, и символьная информация становится устаревшей. При выполнении функции **exec** все точки прерывания удаляются; новая программа останавливается для идентификации, упрощая отладку. Программа **dbx** сама подключается к образу новой программы, вызывает функцию для определения имени новой программы, сообщает это имя, а затем выдает приглашение на ввод. Приглашение выглядит примерно следующим образом:

```
(dbx) multproc
Включен режим параллельной отладки
(dbx) run
Подключение к программе из функции exec . . .
Определение имени программы . . .
Подключение к /home/user/execprog успешно выполнено . . .
Чтение символьной информации . . .
(dbx)
```

Если новый процесс порождается программой с несколькими нитями, то в нем будет существовать только одна нить. Процесс должен вызывать функцию **exec**. В противном случае сохраняется исходная символьная информация, и команды, работающие с нитями (такие как **thread**), будут выдавать информацию об объектах родительского процесса, которая на самом деле уже будет устаревшей. При вызове функции **exec** исходная символьная информация обновляется, и указанные команды будут выдавать информацию об объектах нового дочернего процесса.

Для того чтобы следить за порожденным дочерним процессом, не открывая новое окно Xwindow, укажите в команде **multproc** флаг **child**. В этом случае при порождении процесса программа **dbx** будет следить за дочерним процессом. Если указать в команде **multproc** флаг **parent**, то при порождении процесса программа

dbx остановится, но затем будет отслеживать родительский процесс. Если указать оба флага (и **child**, и **parent**), то будет отслеживаться выполняемый процесс. Эти флаги полезны при отладке программ без запуска Xwindows.

Проверка программных данных

Этот раздел посвящен анализу, проверке и изменению данных в программах.

Обработка сигналов

Программа **dbx** может перехватывать или игнорировать сигналы, отправляемые в вашу программу. Каждый раз, когда в вашу программу отправляется сигнал, программа **dbx** получает об этом уведомление. Если сигнал следует проигнорировать, он передается в вашу программу; в противном случае **dbx** останавливает программу и сообщает о перехвате сигнала. Программа **dbx** не может проигнорировать сигнал **SIGTRAP**, если он поступил от внешнего по отношению к ней процесса. В программе с несколькими нитями сигнал может передаваться в определенную нить с помощью функции **pthread_kill**. По умолчанию программа **dbx** останавливается и сообщает о перехвате сигнала. С помощью команды **ignore** можно указать, что программа **dbx** должна игнорировать сигнал и передавать его нити. Для изменения режима обработки сигнала по умолчанию используются команды **catch** и **ignore**.

В следующем примере сигналы **SIGGRANT** и **SIGREQUEST** применяются программой для выделения ресурсов. Для того чтобы программа **dbx** продолжала работу при получении этих сигналов, введите:

```
(dbx) ignore GRANT
(dbx) ignore SIGREQUEST
(dbx) ignore
CONT CLD ALARM KILL GRANT REQUEST
```

Если установить переменную **$sigblock**, то программа **dbx** будет блокировать сигналы, передаваемые в вашу программу. По умолчанию сигналы, поступающие в программу **dbx**, направляются в исходную программу или в объектный файл, определяемый параметром *Объектный_файл* программы **dbx**. Если с помощью команды **set** установить переменную **$sigblock**, то сигналы, получаемые программой **dbx**, не будут передаваться в исходную программу. Если вы хотите, чтобы сигнал передавался в программу, воспользуйтесь командой **cont** с сигналом в качестве операнда.

Можно использовать этот способ для прерывания выполнения программы, которая запускается программой **dbx**. Обычно перед продолжением выполнения программы проверяется ее состояние. Если переменная **$sigblock** не установлена, то при прерывании выполнения в программу передается сигнал **SIGINT**. При продолжении выполнения программы управление передается обработчику сигнала (если он существует).

Следующий пример программы показывает, как изменяется порядок выполнения при запуске в режиме отладки (т. е. с помощью **dbx**), если установлена переменная **$sigblock**:

```
#include <signal.h>
#include <stdio.h>
void inthand( ) {
        printf("\nПолучен сигнал SIGINT\n");
        exit(0);
}
main( )
{
        signal(SIGINT, inthand);
        while (1) {
                    printf(".");
                  fflush(stdout);
                sleep(1);
        }
}
```

В следующем примере эта программа используется как исходный файл в сеансе работы с программой **dbx**. При первом прогоне программы переменная **$sigblock** не установлена. Во время повторного запуска (rerun) переменная **$sigblock** установлена. Справа в угловых скобках приведены комментарии:

```
dbx версии 3.1.
Введите 'help' для просмотра справки
относительно символьной информации ...
(dbx) run
.........^C                      <Пользователь нажал Ctrl-C здесь!>
interrupt in sleep at 0xd00180bc
0xd00180bc (sleep+0x40) 80410014          1        r2,0x14(r1)
(dbx) cont

SIGINT received

execution completed
(dbx) set $sigblock
(dbx) rerun
[ looper ]
..............^C                     <Пользователь нажал Ctrl-C здесь!>
interrupt in sleep at 0xd00180bc
0xd00180bc (sleep+0x40) 80410014          1        r2,0x14(r1)
(dbx) cont
....^C   <Программа не получила сигнала, выполнение продолжено>

interrupt in sleep at 0xd00180bc
0xd00180bc (sleep+0x40) 80410014          1        r2,0x14(r1)
(dbx) cont 2 <Завершение программы с сигналом 2>

SIGINT received

execution completed
(dbx)
```

Вызов процедур

Из программы **dbx** можно вызывать процедуры для тестирования различных аргументов. Можно также вызывать диагностические процедуры, которые форматируют данные для упрощения отладки. Для вызова процедуры служит команда **call** или команда **print**.

Просмотр данных трассировки стека

Для просмотра списка вызовов процедур, которые предшествовали останову программы, служит команда **where**.

В приведенном ниже примере исполняемый объектный файл **hello** состоит из двух исходных файлов и трех процедур, включая стандартную процедуру `main`. Программа останавливается в точке прерывания в процедуре `sub2`.

```
(dbx) run
[1] останавливается в sub2 в строке 4 в файле "hellosub.c"
(dbx) where
sub2(s = "hello", n = 52), строка 4 в "hellosub.c"
sub(s = "hello", a = -1, k = delete), строка 31 в "hello.c"
main(), строка 19 в "hello.c"
```

При трассировке стека вызовы просматриваются в обратном порядке. Если начать с дна стека, то происходившие события образуют следующую последовательность:

1. Оболочка вызвала процедуру `main`.
2. Процедура `main` вызвала процедуру `sub` в строке 19 со значениями `s = "hello"`, `a = -1` и `k = delete`.
3. Процедура `sub` вызвала процедуру `sub2` в строке 31 со значениями `s = "hello"` и `n = 52`.
4. Программа остановилась на процедуре `sub2` в строке 4.

Часть трассировки стека, начиная с фрейма с номером `0` и до фрейма с номером `1`, можно просмотреть с помощью `where 0 1`.

```
(dbx) run
[1] останавливается в sub2 в строке 4 в файле "hellosub.c"
(dbx) where 0 1
sub2(s = "hello", n = 52), строка 4 в "hellosub.c"
sub(s = "hello", a = -1, k = delete), строка 31 в "hello.c"
```

> **Примечание:** Для того чтобы отключить режим вывода на экран аргументов, передаваемых в процедуры, задайте в программе отладки переменную **$noargs**. Задайте переменную отладки **$stack_details** для отображения номера фрейма и регистра, заданного для каждой активной функции или процедуры.

С помощью команд **up**, **down** и **frame** можно просматривать фрагменты стека.

Просмотр и изменение переменных

Для просмотра выражений применяется команда **print**. Для печати имен и значений переменных - команда **dump**. Если данная процедура - точка, то печатаются все активные переменные. Если задан параметр *PATTERN*, то вместо отображения только указанного символа будут напечатаны все символы, соответствующие *PATTERN*. Для изменения значения переменной предназначена команда **assign**.

В приведенном ниже примере в программе на языке C используются автоматическая целочисленная переменная `x` со значением 7 и параметры `s` и `n` в процедуре `sub2`:

```
(dbx) print x, n
7 52
(dbx) assign x = 3*x
(dbx) print x
21
(dbx) dump
sub2(s = "hello", n = 52)
x = 21
```

Просмотр информации о нитях

Для просмотра информации о пользовательских нитях, взаимных блокировках, условиях и объектах атрибутов используются команды **thread**, **mutex**, **condition** и **attribute**. К этим объектам применима также команда **print**. В следующем примере выполняется нить 1. Пользователь делает текущей нить 2, запрашивает список нитей, печатает информацию о нити 1 и, наконец, печатает информацию о нескольких объектах, связанных с нитями.

```
(dbx) thread current 2
(dbx) thread
 thread  state-k   wchan state-u   k-tid mode held scope function
*$t1     run             running   12755 u    no   pro   main
>$t2     run             running   12501   k   no   sys  thread_1
```

```
(dbx) print $t1
(thread_id = 0x1, state = run, state_u = 0x0, tid = 0x31d3, mode = 0x1, held = 0x0, priority = 0x3c,
    policy = other, scount = 0x1, cursig = 0x5, attributes = 0x200050f8)
```

```
(dbx) print $a1,$c1,$m2
(attr_id = 0x1, type = 0x1, state = 0x1, stacksize = 0x0, detachedstate = 0x0, process_shared = 0x0,
 contentionscope = 0x0, priority = 0x0, sched = 0x0, inherit = 0x0, protocol = 0x0, prio_ceiling = 0x0)
(cv_id = 0x1, lock = 0x0, semaphore_queue = 0x200032a0, attributes = 0x20003628)
(mutex_id = 0x2, islock = 0x0, owner = (nil), flags = 0x1, attributes = 0x200035c8)
```

Область видимости имен

Для преобразования имен в первую очередь применяется статический контекст текущей функции. Динамический контекст используется в том случае, если имя не определено в статическом контексте. Если имя не найдено ни в статическом, ни в динамическом контексте, то выбирается произвольный символ и печатается сообщение `usingQualifiedName`. Можно переопределить процедуру преобразования имен, задав идентификатор и имя блока (например, *Модуль.Переменная*). Исходные файлы рассматриваются как

модули, имена которых совпадают с именами файлов без расширения. Например, полное имя переменной *x*, объявленной в процедуре sub внутри файла **hello.c**, будет **hello.sub.x**. Точка в имени ставится самой программой.

Если есть нескольких идентификаторов одним именем, то для определения того, какой именно символ найден, можно воспользоваться командами **which** и **whereis**.

Использование операторов и модификаторов в выражениях

Программа **dbx** позволяет просматривать широкий диапазон выражений. Для задания выражений используется обычный синтаксис языка C, с некоторыми расширениями языка FORTRAN.

*** (звездочка) или ^ (знак вставки)**
Обозначает косвенную адресацию или разыменование указателя.

[ ] (квадратные скобки) или ( ) (круглые скобки)
Обозначают индексное выражение массива.

. (точка)
Оператор ссылки на поле. Используется в структурах и указателях. При этом оператор языка C -> (стрелка) становится не нужен, хотя и может использоваться.

&(амперсанд)
Получение адреса переменной.

.. (две точки)
Разделяют верхнюю и нижнюю границы при обозначении части массива. Например, **n[1..4]**.

В выражениях допустимы следующие типы операций:

Алгебраические
=, **-**, *****,/(деление вещественных чисел), **div** (деление целых чисел), **mod** (вычисление остатка от деления нацело), **exp** (возведение в степень)

Поразрядные
-, **I**, **bitand**, **xor**, **~**, **<<**, **>>**

Логические
or, **and**, **not**, **II**, **&&**

Сравнение
<, **>**, **<=**, **>=**, **<>**, **!=**, **=** или **==**

Другие
sizeof

Логические операции и операции сравнения разрешается использоваться для записи условий в командах **stop** и **trace**.

Контроль типов выражений

Программа **dbx** контролирует типы выражений. Вы можете переопределять тип выражения с помощью оператора переименования или приведения типа. Существуют три способа переименования типа:

- *Тип* **(***выражение***)**
- *Выражение* **** *Имя-типа*
- **(***Имя-типа***)***Выражение*

> **Примечание:** Если исходным или целевым типом при приведении является структура, объединение или класс, то выполняется выравнивание влево. Однако при приведении класса к базовому классу соблюдаются синтаксические правила языка C++.

Например, для переименования переменной x целого типа со значением 97 введите:

```
(dbx) print char (x), x \ char, (char) x, x,
'a' 'a' 'a' 97
```

В следующем примере проиллюстрировано приведение типов с помощью формата **(***имя-типа***)** *выражение*:

```
print (float) i
print ((struct qq *) void_pointer)->first_element
```

Преобразование типов в программе **dbx** выполняется по правилам языка C со следующими ограничениями:

- Не поддерживаются в качестве операторов приведения типы, используемые в языке Fortran (integer*1, integer*2, integer*4, logical*1, logical*2, logical*4 и т. д.).
- Если активная переменная имеет то же имя, что и один из базовых или пользовательских типов, то этот тип не может применяться в качестве оператора преобразования типов языка C.

Команда **whatis** выдает описание идентификатора, который может быть уточнен именем блока.

Конструкция **$$***тег* применяется для печати описаний тегов перечислимых типов данных, структур и объединений/

Тип выражения в команде **assign** должен совпадать с типом переменной, который вы присваиваете. Если типы не совпадают, будет выдано сообщение об ошибке. Вы можете изменить тип выражения с помощью операции переименования типа. Для отключения контроля типов укажите специальную переменную **$unsafeassign** программы **dbx**.

Перевод переменных в нижний или верхний регистр

По умолчанию программа **dbx** выполняет преобразование символов к нужному регистру в соответствии с текущим языком. Если текущий язык - C, C++ или не определен, то регистр символов не изменяется. Если текущий язык - Fortran, то символы переводятся в нижний регистр. Текущий язык остается неопределенным, если программа находится в разделе кода, который не компилировался с флагом **debug**. Изменить настройку, установленную по умолчанию, можно с помощью команды **case**.

Команда **case** без аргументов показывает текущий регистр.

Компилятор Fortran преобразует все символы программы в нижний регистр; компилятор C - нет. Однако некоторые компиляторы Fortran не всегда изменяют регистр символов на нижний. Например, для процедуры **proc1**, входящей в состав модуля **mod2** компилятор XLF Fortran создаст имя **__mod2_MOD_proc1**, содержащее символы смешанного регистра. В таких случаях в программе **dbx** следует разрешить символы **смешанного регистра**.

Изменение формата вывода с помощью специальных переменных программы отладки

С помощью команды **set** можно задать специальные переменные программы **dbx**, которые изменяют формат вывода команды **print**:

$hexints
Печатает целые выражения в шестнадцатеричном формате.

$hexchars
Печатает символьные выражения в шестнадцатеричном формате.

$hexstrings
Печатает адреса строк символов, а не сами строки.

$octints
Печатает целые выражения в восьмеричном формате.

$expandunions
Печатает поля в объединении.

$pretty
Печатает сложные типы C и C++ в формате **pretty**.

$print_dynamic
Включает вывод динамического типа объектов C++.

$show_vft
Включает вывод таблицы виртуальных функций во время вывода объектов C++.

Для получения результатов отладки в нужном вам формате установите необходимые переменные программы отладки. Пример:

```
(dbx) whatis x; whatis i; whatis s
int x;
char i;
char *s;
(dbx) print x, i, s
375 'c' "hello"
(dbx) set $hexstrings; set $hexints; set $hexchars
(dbx) print x, i, s
0x177 0x63 0x3fffe460
(dbx) unset $hexchars; set $octints
(dbx) print x, i
0567 'c'
(dbx) whatis p
struct info p;
(dbx) whatis struct info
struct info {
    int x;
    double position[3];
    unsigned char c;
    struct vector force;
};
(dbx) whatis struct vector
struct vector {
    int a;
    int b;
    int c;
};
(dbx) print p
(x = 4, position = (1.3262493258532527e-315, 0.0, 0.0),
c = '\0', force = (a = 0, b = 9, c = 1))(dbx) set $pretty="on"
(dbx) print p
{
    x = 4
    position[0] = 1.3262493258532527e-315
    position[1] = 0.0
    position[2] = 0.0
    c = '\0'
    force = {
        a = 0
        b = 9
        c = 1
    }
}
(dbx) set $pretty="verbose"
(dbx) print p
x = 4
position[0] = 1.3262493258532527e-315
position[1] = 0.0
position[2] = 0.0
```

```
c = '\0'
force.a = 0
force.b = 9
force.c = 1
```

Когда show_vft выключен, и объект выводится командой sub, содержимое таблицы виртуальных функций (VFT) не выводится. Если этот параметр включен, данные VFT выводятся. Пример:

```
(dbx) p *d
    B1:(int_in_b1 = 91)
    B2:(int_in_b2 = 92)
(int_in_d = 93)
(dbx) p *b2
(int_in_b2 = 20)
(dbx)set $show_vft
(dbx) p *d
    B1:(B1::f1(), int_in_b1 = 91)
    B2:(D::f2(), int_in_b2 = 92)
(int_in_d = 93)
(dbx) p *b2
(B2::f2(), int_in_b2 = 20)
(dbx)
```

Когда print_dynamic выключен, объект выводится по своему статическому типу (который указан в исходном коде). В противном случае, объект выводится по своему динамическому типу (который объект имел до выполнения преобразования типа). Пример:

```
(dbx) r
[1] остановлен в функции main, строка 57
   57   A *obj1 = new A();
(dbx) n
остановлен в функции main, строка 58
   58   A *obj2 = new B();
(dbx) n
остановлен в функции main, строка 59
   59 cout<<" a = "<<obj2->a<<" b = "<<obj2->b<<endl;
(dbx) p *obj2
(a = 1, b = 2)
(dbx)set $print_dynamic
(dbx) print *obj2
    A:(a = 1, b = 2)
(c = 3, d = 4)
(dbx)
```

Применение dbx для отладки на машинном уровне

Программу **dbx** можно применять для отладки программ на уровне ассемблера. Можно просматривать и изменять адреса памяти, просматривать команды ассемблера и пошаговые инструкции, устанавливать точки прерывания и события трассировки по адресам памяти и просматривать регистры.

В приведенных ниже командах и примерах адрес - это выражение, значение которого равно адресу памяти. В наиболее общем виде адрес задается как целое число или как выражение, в котором вычисляется адрес идентификатора с помощью оператора **&** (амперсанд). В командах машинного уровня можно также задавать адрес в виде выражения, заключенного в круглые скобки. Адреса могут состоять из других адресов и операторов сложения (**+**), вычитания (**-**) и косвенной адресации (унарная операция ***).

Дополнительная информация об отладке программ на машинном уровне с помощью **dbx** приведена в следующих разделах.

Работа с машинными регистрами

Для просмотра содержимого машинных регистров предназначена команда **registers**. Регистры делятся на три группы: регистры общего назначения, регистры с плавающей точкой и регистры управления системой.

Регистры общего назначения

Регистры общего назначения обозначаются **$r***номер*, где *номер* - номер регистра.

> **Примечание:** Значение регистра может быть равно шестнадцатеричному числу `0xdeadbeef`. Это значение присваивается всем регистрам общего назначения при инициализации.

Регистры с плавающей точкой

Регистры с плавающей точкой обозначаются **$fr***номер*, где *номер* - это номер регистра. По умолчанию содержимое регистров с плавающей точкой не выводится. Для просмотра регистров с плавающей точкой необходимо сбросить переменную **$noflregs** программы отладки (**unset $noflregs**). Кроме того, регистры с плавающей - точкой можно указать в командах **print** и **assign** по типам. **$frNumber** значения по умолчанию для двойного типа. **$frNumberh** ссылается на регистр с плавающей точкой как тип **_Decimal32**. **$frNumberd** ссылается на регистр с плавающей точкой как тип **_Decimal64**. Ниже приведены примеры различных типов регистров с плавающей точкой:

```
(dbx) print $fr0

1.10000002

(dbx) print $fr0h

1.100001

(dbx) print $fr0d

1.10000062

(dbx) assign $fr0 = 9.876

(dbx) assign $fr0h = 9.876df

(dbx) assign $fr0d = 9.876dd
```

Векторные регистры

Векторные регистры обозначаются $vr*номер*, где *номер* - это номер регистра. Векторные регистры не отображаются по умолчанию и присутствуют только в процессорах, поддерживающих блок обработки векторов (VPU).

Для просмотра векторных регистров необходимо сбросить переменную **$novregs** программы отладки (`unset $novregs`). Кроме того, векторные регистры можно указать в командах **print** и **assign** по типам. По умолчанию для $vr*номер* указывается тип вектора int. $vr*номер*f указывает на вектор типа float. $vr*номер*s указывает на вектор типа short. $vr*номер*c указывает на вектор типа char.

Ниже приведены примеры различных типов векторных регистров:

```
(dbx) print $vr20

((1066192077, 1074161254, 1078355558, 1082340147))

(dbx) print $vr20f

((1.10000002, 2.0999999, 3.0999999, 4.0999999))
```

```
(dbx) print $vr20s

((16268, 52429, 16390, 26214, 16454, 26214, 16515, 13107))

(dbx) assign $vr20f[3] = 9.876

(dbx) print $vr20f ((1.10000002, 2.0999999, 3.0999999, 9.8760004))
```

Регистры управления системой

Поддерживаются следующие регистры управления системой:

- Регистр адреса команды: **$iar** или **$pc**
- Регистр состояния условия: **$cr**
- Регистр множителя-частного: **$mq**
- Регистр машинного состояния: **$msr**
- Регистр ссылки: **$link**
- Регистр счетчика: **$ctr**
- Регистр исключительной ситуации при операциях с фиксированной точкой: **$xer**
- Регистр ИД транзакции: **$tid**
- Регистр состояния операций с плавающей точкой: **$fpscr**

Проверка адресов памяти

С помощью приведенной ниже команды можно просматривать содержимое регистров (для этого необходимо указать начальный адрес и либо конечный адрес, либо число элементов в переменной *Счетчик*). Переменная *Режим* определяет способ представления содержимого памяти при печати.

Адрес, *Адрес* / **[***Режим***][>** *Файл***]**

Адрес / **[***Счетчик***][***Режим***] [>** *Файл***]**

Если переменная *Режим* не задана, используется предыдущий режим. Начальный режим - **X**. Поддерживаются следующие режимы печати:

b Печать байта в восьмеричном формате.

c Печать байта в виде символа.

Д Печать длинного слова в десятичном формате.

d Печать короткого слова в десятичном формате.

Df Печать числа с десятичной плавающей точкой с двойной точностью.

DDf
Печать числа с десятичной плавающей точкой с четверной точностью.

f Печать числа с плавающей точкой с одинарной (обычной) точностью.

g Печать числа с плавающей точкой с двойной точностью.

Hf Печать числа с десятичной плавающей точкой с одинарной точностью.

h Печать байта в шестнадцатеричном формате.

i Печать машинной команды.

lld
Печать 8-байтового десятичного числа со знаком.

ll o
Печать 8-байтового восьмеричного числа без знака.

ll u
Печать 8-байтового десятичного числа без знака.

ll x
Печать 8-байтового шестнадцатеричного числа без знака.

0 Печать длинного слова в восьмеричном формате.

o Печать короткого слова в восьмеричном формате.

q Печать числа с плавающей точкой с расширенной точностью.

s Печать строки символов, оканчивающейся символом NULL.

X Печать длинного слова в шестнадцатеричном формате.

x Печать короткого слова в шестнадцатеричном формате.

В приведенном ниже примере выражения в круглых скобках могут использоваться как адреса:

```
(dbx) print &x
0x3fffe460
(dbx) &x/X
3fffe460: 31323300
(dbx) &x,&x+12/x
3fffe460: 3132 3300 7879 7a5a 5958 5756 003d 0032
(dbx) ($pc)/2i
100002cc (sub)  7c0802a6        mflr     r0
100002d0 (sub + 0x4)    bfc1fff8        stm       r30,-8(r1)
```

Выполнение программы на машинном уровне

Для отладки программы на машинном уровне используются почти те же команды, что и для отладки на символьном уровне. Команда **stopi** останавливает процессор при достижении определенного адреса, выполнении условия или изменении переменной. Команды **tracei** работают так же, как и команды символьной трассировки. Команда **stepi** выполняет одну команду или указанное *Число* машинных команд.

Если бы вы выполнили другую команду **stepi** в этой точке, то вы бы остановились на адресе 0x10000618, обозначенном как точка входа для процедуры `printf`. Если вы не планируете останавливаться на этом адресе, вы могли бы использовать команду **return**, чтобы продолжить выполнение со следующей команды в `sub` с адресом 0x100002e0. В этой точке команда **nexti** будет автоматически продолжать выполнение до адреса 0x10000428.

Если в вашей программе существует несколько нитей, то при останове программы на экран будет выдано символьное имя работающей нити. Например:

```
останов в sub по адресу 0x100002d4 ($t4)
10000424 (sub+0x4) 480001f5 bl 0x10000618 (printf)
```

Отладка исполняемых файлов, переупорядоченных с помощью fdpr

Вы можете отлаживать программы, которые были переупорядочены на уровне команд с помощью команды **fdpr** (реструктурирование направленной программы с обратной связью, компонент Performance Toolbox for AIX). Если применяются опции оптимизации **-R0** или **-R2**, то **dbx** может выдавать информацию о соответствии между преобразованными адресами команд и адресами в исходном исполняемом файле:

```
0xRRRRRRRR = fdpr[0xYYYYYYYY]
```

В данном примере `0xRRRRRRRR` - переупорядоченный адрес, а `0xYYYYYYYY` - исходный адрес. Кроме того, **dbx** использует записи обратной трассировки в исходной области команд для поиска связанных имен процедур для сообщения `stopped in`, команды **func** и обратной трассировки.

```
(dbx) stepi
stopped in proc_d at 0x1000061c = fdpr[0x10000278]
0x1000061c (???) 9421ffc0       stwu   r1,-64(r1)
(dbx)
```

В предыдущем примере **dbx** указывает, что выполнение программы было остановлено в функции `proc_d` с адресом `0x1000061c` в переупорядоченном текстовом разделе, который первоначально располагался по адресу `0x10000278`. Дополнительная информация о **fdpr** содержится в описании команды **fdpr**.

Просмотр команд ассемблера

Для просмотра набора команд из исходного файла предназначена команда **listi** программы **dbx**. В режиме по умолчанию программа **dbx** выдает список команд для архитектуры, в которой она работает. Можно переопределить режим по умолчанию с помощью переменных **$instructionset** и **$mnemonics** команды **dbx set**.

Дополнительная информация о просмотре и дезассемблировании команд приведена в описании команды **listi** программы **dbx**. Дополнительная информация о переопределении режима по умолчанию приведена в описании переменных **$instructionset** и **$mnemonics** команды **dbx set**.

Настройка среды отладки dbx

Вы можете настраивать среду отладки, создавая псевдонимы команд и задавая опции в файле **.dbxinit**. Команды **dbx** можно считывать из файла с помощью флага **-c**.

Дополнительная информация об опциях настройки содержится в следующих разделах:

Определение нового приглашения dbx

Обычно приглашение **dbx** - это имя, используемое для запуска программы **dbx**. Если в командной строке вы указали `/usr/ucb/dbx a.out`, то приглашение будет иметь вид `/usr/ucb/dbx`.

Вид приглашения можно изменить либо с помощью команды **prompt**, либо задав другое приглашение в строке **prompt** файла **.dbxinit**. Изменение приглашения в файле **.dbxinit** приводит к тому, что каждый раз при инициализации программы **dbx** вместо приглашения по умолчанию будет выдаваться заданное вами приглашение.

Например, для того чтобы при инициализации программы **dbx** выдавалось приглашение в виде `debug->`, укажите в файле **.dbxinit** следующую строку:

```
prompt "debug-->"
```

Создание псевдонимов команд dbx

Вы можете создавать свои собственные команды на основе набора базовых команд **dbx**. Следующие команды позволяют создавать пользовательский псевдоним из указанных аргументов. Все команды в строке замещения для псевдонима должны быть базовыми командами **dbx**. В дальнейшем вы можете использовать свои псевдонимы вместо базовых команд **dbx**.

Команда **alias** без аргументов выдает список действующих псевдонимов; эта же команда с одним аргументом выдает строку замещения, связанную с данным псевдонимом.

alias [*псевдоним*[*команда*]]

alias*псевдоним "команда"*

alias *псевдоним* **(***параметр-1, параметр-2 . . .* **)** *"команда"*

Первые две формы команды **alias** предназначены для подстановки строки замещения вместо псевдонима. Третья форма - это ограниченное средство макроязыка. В строку замещения подставляется каждый параметр, указанный в команде **alias**.

Ниже перечислены псевдонимы по умолчанию и соответствующие им команды:

attr
attribute

bfth
stop (в данной нити в указанной функции)

blth
stop (в данной нити в указанной строке исходного файла)

c cont

cv condition

d delete

e edit

h help

j состояние

l list

m map

mu взаимная блокировка

n next

p **print**

q quit

r run

s step

st stop

t **where**

th thread

x registers

Для удаления (отмены) псевдонима предназначена команда **unalias**.

Работа с файлом .dbxinit

При запуске сеанса отладки программа **dbx** пытается найти специальные файлы инициализации с именами **.dbxinit**, в которых хранится список выполняемых команд **dbx**. Эти команды выполняются до того, как программа **dbx** начнет обрабатывать команды стандартного ввода. Сразу после запуска команда **dbx** ищет файл **.dbxinit** в текущем пользовательском каталоге или в пользовательском каталоге **$HOME**. Если файл **.dbxinit** существует, то в начале сеанса отладки выполняются команды из этого файла. Если файл **.dbxinit** есть и в домашнем, и в текущем каталоге, то оба файла считываются в указанном порядке. Так как файл **.dbxinit** из текущего каталога считывается вторым, то команды из этого файла могут изменить результат выполнения команд из файла, расположенного в домашнем каталоге.

Как правило, в файле **.dbxinit** указываются команды **alias**, но можно задавать и другие команды **dbx**. Пример:

```
$ cat .dbxinit
alias si "stop in"
prompt "dbg-->"
$ dbx a.out
dbx, версия 3.1
Введите 'help' для получения справки.
чтение символьной информации . . .
dbg--> alias
si   stop in
t    where . . .
dbg-->
```

считывание команд dbx из файла

Для выполнения команд **dbx** до начала сеанса отладки применяется флаг **-c** и файл **.dbxinit**. Если указать флаг **-c**, то программа **dbx** не будет искать файл **.dbxinit**. В этом случае после начала сеанса отладки для чтения команд **dbx** из файла следует воспользоваться командой **source**.

После выполнения команд из файла **cmdfile** программа **dbx** выдает приглашение и ожидает ввода.

Для определения списка команд, которые должны выполняться при начальном запуске программы **dbx**, можно также воспользоваться опцией **-c**.

Отладка циклических блокировок

С помощью программы **dbx** можно проводить отладку циклических блокировок. Для этого нужно присвоить переменной среды AIXTHREAD_SPINLOCKS значение ON.

Среда разработки встраиваемых модулей dbx

В программе **dbx** предусмотрена среда разработки встраиваемых модулей, позволяющая добавлять в **dbx** новые команды и обработчики событий.

Каждый пользователь dbx может создать встраиваемый модуль, расширяющий **dbx** вспомогательными командами, связанными с приложением или библиотекой.

Примечание:

1. Поскольку команда **dbx** является 64-разрядным процессом, все встраиваемые модули, используемые с командой **dbx**, необходимо компилировать для 64-разрядного режима процессора. Для загрузки 32-разрядных встраиваемых модулей требуется 32-разрядная версия команды **dbx**, она называется **dbx32**.
2. В процессе разработки следует отличать **процедуры обратного вызова dbx** и **процедуры интерфейса встраиваемых модулей**.
3. **Процедуры обратного вызова dbx** - это набор служб **dbx**, доступных для встраиваемого модуля. Доступ к этим процедурам встраиваемому модулю предоставляется с помощью указателей на функции.
4. **Процедуры интерфейса встраиваемых модулей** - это набор методов **dbx**, которые должны быть реализованы встраиваемым модулем.

Формат файла

Каждый встраиваемый модуль должен представлять собой общий объектный файл.

Присвоение имен

Перенаправление ввода команд **dbx** выполняется в соответствии с *уникальными именами* встраиваемых модулей.

Программа **dbx** извлекает уникальное имя из имени файла встраиваемого модуля. В ходе инициализации **dbx** выполняет поиск файлов, имена которых соответствуют следующему регулярному выражению, в стандартных и пользовательских каталогах:

```
^libdbx_.+\.so$
```

В следующей таблице перечислены примеры допустимых и недопустимых имен файлов для **встраиваемых модулей dbx**. Для каждого допустимого примера указано соответствующее *уникальное имя*:

Имя файла	Допустимое	Уникальное имя
libdbx_sample.so	Yes	sample
libdbx_xyz.so	Yes	xyz
libdbx_my_app.so	Yes	my_app
libdbx.so	No	
libdbx_.so	No	
libdbx_sample.so.plugin	No	
plugin_libdbx_sample.so	No	

Источник информации

Программа **dbx** позволяет указать список каталогов для поиска с помощью переменной среды DBX_PLUGIN_PATH. Для разделения каталогов в списке применяется двоеточие. В следующем примере двоеточие разделяет два каталога.

```
$ export dbx_PLUGIN_PATH=$HOME/dbx_plugins:/mnt/share/dbx_plugins
```

Программа **dbx** выполняет поиск встраиваемых модулей в ходе инициализации. **dbx** просматривает каталог исполняемых файлов (если он указан). Поиск по этому каталогу выполняется после просмотра пользовательских каталогов.

Примечание: Если **dbx** применяется для прикрепления процесса, полный путь к исполняемому файлу определить нельзя.

Загрузка

Загрузка встраиваемого модуля может быть выполнена одним из следующих способов:

- Автоматическая загрузка и инициализация встраиваемого модуля, если он расположен в одном из каталогов, просматриваемых программой **dbx**. В этом случае загрузка выполняется в ходе инициализации **dbx**.
- Загрузка и инициализация встраиваемого модуля вручную путем указания его расположения в подкоманде **pluginload dbx**. Это можно сделать в любой момент в сеансе работы с **dbx**.

После успешной загрузки встраиваемого модуля в автоматическом режиме или вручную появляется сообщение, аналогичное следующему:

```
(dbx) pluginload /home/user/dbx_plugins/libdbx_sample.so
 встраиваемый модуль "/home/user/dbx_plugins/libdbx_sample.so" загружен
```

Если имя загружаемого встраиваемого модуля совпадает с *уникальным именем* уже загруженного встраиваемого модуля, то загрузка отменяется и выдается предупреждающее сообщение, аналогичное следующему.

```
(dbx) pluginload /mnt/share/dbx_plugins/libdbx_sample.so

не удалось загрузить встраиваемый модуль
"/mnt/share/dbx_plugins/libdbx_sample.so":
встраиваемый модуль "/home/user/dbx_plugins/libdbx_sample.so" уже загружен.
```

Выгрузка

Указав имя встраиваемого модуля в команде **pluginunload dbx**, вы можете вручную выгрузить его. При этом способ загрузки встраиваемого модуля значения не имеет. После успешной выгрузки встраиваемого модуля появляется сообщение, аналогичное следующему.

```
(dbx) pluginunload sample
встраиваемый модуль "/home/user/dbx_plugins/libdbx_sample.so" выгружен.
```

Управление версиями

В случае внесения изменений в среду разработки для сохранения совместимости с существующими модулями предыдущих версий создается новый идентификатор версии. Это касается всех значительных изменений или добавлений, вносимых в **Интерфейс встраиваемых модулей** и **Процедуры обратного вызова встраиваемых модулей dbx**.

Для того чтобы максимально уменьшить частоту изменения версий встраиваемых модулей, в некоторых **процедурах обратного вызова dbx** предусмотрен дополнительный параметр, указывающий размер буфера. Такой подход применяется для параметров буферов, основанных на структурах системы, размер которых не управляется **dbx**. В этом случае размер структур системы можно изменять без обновления версии встраиваемого модуля.

На данный момент единственным идентификатором версии является DBX_PLUGIN_VERSION_1.

Заголовочный файл

Прототипы функций, определения структур данных, а также определения макрокоманд, предназначенные для разработки **встраиваемых модулей dbx**, расположены в следующем заголовочном файле:

```
/usr/include/sys/dbx_plugin.h
```

Интерфейс встраиваемых модулей

Прототипы и определения процедур **интерфейса встраиваемых модулей** расположены в заголовочном файле `dbx_plugin.h`.

Каждый встраиваемый модуль *должен* реализовать и экспортировать все следующие процедуры:

- `int dbx_plugin_version(void)`
- `int dbx_plugin_session_init(dbx_plugin_session_t session, constdbx_plugin_service_t *servicep)`
- `void dbx_plugin_session_command(dbx_plugin_session_t session, int argc, char *const argv[])`
- `void dbx_plugin_session_event(dbx_plugin_session_t session, int event, dbx_plugin_event_info_t *event_infop)`

int dbx_plugin_version(void)

Эта процедура возвращает идентификатор версии **встраиваемого модуля dbx**, соответствующий версии встраиваемого модуля. На данный момент единственным идентификатором версии является `DBX_PLUGIN_VERSION_1`.

int dbx_plugin_session_init(dbx_plugin_session_t session, constdbx_plugin_service_t *servicep)

Эта процедура выполняет инициализацию встраиваемого модуля, необходимую для его правильной работы, перед передачей управления обратно программе **dbx**. При необходимости на данном этапе настраиваются псевдонимы команд встраиваемого модуля.

В результате выполнения этой процедуры создается сеанс встраиваемого модуля, связывающий указанный идентификатор сеанса с прикладной программой или файлом дампа. Для идентификации процесса или файла дампа в вызовах **интерфейса встраиваемых модулей** и запросах **процедур обратного вызова dbx** программа **dbx** и встраиваемый модуль применяют идентификатор сеанса. Кроме того, данная процедура поддерживает структуры процедур обратного вызова.

Если инициализация выполнена успешно, эта процедура возвращает нулевое значение. В противном случае, **dbx** выгружает и отменяет встраиваемый модуль.

void dbx_plugin_session_command(dbx_plugin_session_t session, int argc, char *const argv[])

Данная процедура принимает входные данные от пользователя **dbx**, указанные в виде аргументов команды **plugin**. Синтаксис команды **plugin**:

`plugin` *`имя`* [*`арг-0 арг-1 арг-2 ... арг-n`*]

Такой подход позволяет пользователю **dbx** указать для каждого встраиваемого модуля любые входные данные. Встраиваемый модуль может выполнить дополнительную фильтрацию входных данных.

Команда **plugin** передает команду, указанную в параметрах *arg**, встраиваемому модулю, имя которого указано в параметре *имя*. (Например, встраиваемый модуль может называться `libdbx_`*`имя`*`.so`) С помощью данной процедуры **dbx** передает встраиваемому модулю параметры *arg0 - argn*. *Значение argv[0]* соответствует *arg0*, *argv[1] - arg1* и так далее.

В большинстве случаев параметр *arg0* содержит имя команды, определенной встраиваемым модулем, а параметры *arg1 - argn* содержат дополнительные флаги и аргументы. Однако такой порядок не обязателен.

Разработчикам рекомендуется создавать команду **help**, выдающую справочную информацию о встраиваемом модуле.

void dbx_plugin_session_event(dbx_plugin_session_t session, int event, dbx_plugin_event_info_t *event_infop)

Данная процедура должна выполнять действия, заданные встраиваемым модулем, в ответ на события прикладной программы. Программа **dbx** вызывает ее каждый раз при возникновении соответствующего события. В следующей таблице перечислены типы событий, уведомления о которых отправляются встраиваемому модулю:

ИД (событие)	Связанные данные (event_infop)	Причина
DBX_PLUGIN_EVENT_RESTART	Нет	Пользователь **dbx** выполнил команду **run**.
DBX_PLUGIN_EVENT _EXIT	Код выхода	Работа прикладной программы завершена с помощью процедуры exit.
DBX_PLUGIN_EVENT _TERM	Номер сигнала завершения	Работа прикладной программы завершена в результате получения необработанного сигнала.
DBX_PLUGIN_EVENT _LOAD	структура dbx_plugin_modinfo_t загруженного модуля	Модуль загружен в прикладную программу.
DBX_PLUGIN_EVENT _UNLOAD	структура dbx_plugin_modinfo_t выгруженного модуля	Модуль выгружен из прикладной программы.

ИД (событие)	Связанные данные (event_infop)	Причина
DBX_PLUGIN_EVENT_BP	Нет	Работа прикладной программы приостановлена в результате обнаружения пользовательской или внутренней точки прерывания **dbx** или точки отслеживания данных.
DBX_PLUGIN_EVENT_SIGNAL	Номер сигнала	Работа прикладной программы приостановлена в результате доставки сигнала.
DBX_PLUGIN_EVENT_SWTHRD	Описатель текущей нити pthread	Выполнение команды **thread current<handle>** пользователем **dbx** привело к изменению текущей нити pthread.

События DBX_PLUGIN_EVENT_BP и DBX_PLUGIN_EVENT_SIGNAL создаются в результате остановки работы программы. Они указывают на то, что данные, сохраненные в кэше и обрабатываемые встраиваемым модулем, могут быть недопустимы. В случае получения уведомления об этих событиях аннулирование данных, сохраненных в кэше, более эффективно, чем их обновление. Полное обновление данных, сохраненных в кэше, следует выполнять только в том случае, если эти данные необходимы. Это в особенности важно, так как **dbx** может игнорировать некоторые сигналы, а некоторые точки прерывания могут быть внутренними точками прерывания. Если пользователь не имеет возможности выполнить команды перед возобновлением работы приложения, то повторное обновление данных приводит к бесполезной трате ресурсов.

```
void dbx_plugin_session_destroy(dbx_plugin_session_t session)
```

Данная процедура должна выполнять обязательные задачи по очистке и управлению памятью.

Процедуры обратного вызова dbx

Ниже перечислены **процедуры обратного вызова dbx**, доступные для каждого встраиваемого модуля с помощью процедуры `dbx_plugin_session_init`.

Процедура обратного вызова dbx **session** позволяет получить характеристики сеанса dbx. Программа **dbx** задает параметр *flagsp*.

```
typedef int (*dbx_plugin_session_service_t)(dbx_plugin_session_t session,
                                            dbx_plugin_session_flags_t *flagsp).
```

Ниже перечислены параметры процедуры обратного вызова dbx **session**:

Параметр
Описание

session
Идентификатор сеанса.

flagsp
Любая комбинация следующих характеристик сеанса:

- DBX_PLUGIN_SESSION_64BIT

  Если задан, то сеанс представляет 64-разрядную прикладную программу. В противном случае, сеанс представляет 32-разрядную программу.
- DBX_PLUGIN_SESSION_CORE

  Если задан, то сеанс представляет файл дампа. В противном случае, сеанс представляет текущий процесс.

Ниже перечислены коды возврата процедуры обратного вызова dbx **session**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимый сеанс
- DBX_PLUGIN_BAD_POINTER - значение *flagsp* равно NULL

процесс

Процедура обратного вызова dbx **process** позволяет получить информацию об отлаживаемом процессе. Программа **dbx** задает параметр *infop*.

```
typedef int (*dbx_plugin_process_service_t)(dbx_plugin_session_t session,
                                            dbx _plugin_procinfo_t *infop,
                                            size_t procinfo_size)
```

Ниже перечислены параметры процедуры обратного вызова dbx **process**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенная структура dbx_plugin_procinfo_t

procinfo_size
Размер структуры dbx_plugin_procinfo_t

Ниже перечислены коды возврата процедуры обратного вызова dbx **process**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *infop* равно NULL
- DBX_PLUGIN_BAD_ARG - недопустимое значение *procinfo_size*
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо информация отсутствует в дампе

fds

Процедура обратного вызова dbx **fds** позволяет получить информацию о дескрипторах файла процесса. Существует два способа получения этой информации:

- Последовательно вызывая эту процедуру, получить информацию о каждом дескрипторе файла. Или
- Вызвать ее один раз для получения общего количества дескрипторов файла и еще раз для получения информации сразу о всех дескрипторах файла.

Если встраиваемый модуль передает ненулевой буфер *infop*, то программа **dbx** загружает в него записи, число которых указано в параметре **countp*, начиная с нити, указанной в параметре **indexp*.

Если встраиваемый модуль в параметре **countp* передает число, превышающее число оставшихся записей, то программа **dbx** загружает все оставшиеся записи. Программа **dbx** обновляет значение *countp* в соответствии с фактическим числом записанных записей, а *indexp* в соответствии с индексом следующего модуля. В случае загрузки последнего дескриптора файла в параметре *indexp* указывается значение -1. Если встраиваемый модуль передает нулевой буфер infop, то значения *indexp* и *countp* все равно обновляются, как если бы *infop* не был нулевым.

```
typedef int (*dbx_plugin_fds_service_t)(dbx_plugin_session_t session,
                                        dbx_plugin_fdinfo_t *infop,
                                        size_t fdinfo_size,
                                        unsigned int *indexp,
                                        unsigned int *countp)
```

Ниже перечислены параметры процедуры обратного вызова dbx **fds**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенный массив структур dbx_plugin_fdinfo_t или NULL

fdinfo_size
Размер отдельной структуры dbx_plugin_fdinfo_t

indexp
Начальный или следующий дескриптор файла (нулевое значение соответствует первому дескриптору файла)

countp
Число дескрипторов файла

Ниже перечислены коды возврата процедуры обратного вызова dbx **fds**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *indexp* или countp равно NULL
- DBX_PLUGIN_BAD_ARG - недопустимое значение *fdinfo_size* или * *countp* == 0
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо информация отсутствует в дампе

modules

Процедура обратного вызова dbx **modules** позволяет получить информацию о загруженных модулях процесса. Существует два способа получения этой информации:

- Последовательно вызывая эту процедуру, получить информацию о каждом модуле. Или
- Вызвать ее один раз для получения общего количества модулей и еще раз для получения информации сразу о всех модулях.

Если встраиваемый модуль передает ненулевой буфер infop, то программа **dbx** загружает в него записи, число которых указано в параметре **countp*, начиная с модуля, указанного в параметре **indexp*.

Если встраиваемый модуль в параметре **countp* передает число, превышающее число оставшихся записей, то программа **dbx** загружает все оставшиеся записи. Программа **dbx** обновляет значение *countp* в соответствии с фактическим числом записанных записей, а *indexp* - в соответствии с индексом следующего модуля. В случае загрузки последнего модуля в параметре *indexp* указывается значение `-1`. Если встраиваемый модуль передает нулевой буфер infop, то значения *indexp* и *countp* все равно обновляются, как если бы *infop* не был нулевым.

Примечание: Данная процедура выделяет память для сохранения имени файла и соответствующих строк символов. После завершения работы инициатор должен освободить эту память.

```
typedef int (*dbx_plugin_modules_service_t)(dbx_plugin_session_t session,
                                            dbx_plugin_modinfo_t *infop,
                                            size_t modinfo_size,
                                            unsigned int *indexp,
                                            unsigned int *countp)
```

Ниже перечислены параметры процедуры обратного вызова dbx **modules**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенный массив структур dbx_plugin_modinfo_t или NULL

modinfo_size
Размер отдельной структуры dbx_plugin_modinfo_t

indexp
Начальный или следующий модуль (нулевое значение соответствует первому модулю)

countp
Число модулей

Ниже перечислены коды возврата процедуры обратного вызова dbx **modules**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *indexp* или countp равно NULL
- DBX_PLUGIN_BAD_ARG - недопустимое значение *modinfo_size* или **countp* == 0

regions

Процедура обратного вызова dbx **regions** позволяет получить информацию о различных областях памяти процесса.

Извлекаются следующие области памяти:

- Стек главной нити (DBX_PLUGIN_REGION_STACK)
- Область пользовательских данных (DBX_PLUGIN_REGION_DATA)
- Область личных данных процесса (DBX_PLUGIN_REGION_SDATA)
- Область преобразования памяти (DBX_PLUGIN_REGION_MMAP)
- Область общей памяти (DBX_PLUGIN_REGION_SHM)

Существует два способа получения этой информации:

- Последовательно вызывая эту процедуру, получить информацию о каждой области. Или
- Вызвать ее один раз для получения общего числа областей и еще раз для получения информации сразу о всех областях.

Если встраиваемый модуль передает ненулевой буфер *infop*, то программа **dbx** загружает в него записи, число которых указано в параметре **countp*, начиная с области, указанной в параметре **indexp*.

Если встраиваемый модуль в параметре **countp* передает число, превышающее число оставшихся записей, то программа **dbx** загружает все оставшиеся записи. Программа **dbx** обновляет значение *countp* в соответствии с фактическим числом записанных записей, а *indexp* - в соответствии с индексом следующей области.

В случае загрузки последней области в параметре *indexp* указывается значение -1. Если встраиваемый модуль передает нулевой буфер *infop*, то значения *indexp* и *countp* все равно обновляются, как если бы *infop* не был нулевым.

Примечание: В настоящий момент данная процедура реализована только для сеансов, представляющих файлы дампов. Для сеансов, представляющих текущие процессы, достаточная информация недоступна программе **dbx**. Вызов этой процедуры для таких сеансов вернет значение DBX_PLUGIN_UNAVAILABLE.

```
typedef int (*dbx_plugin_regions_service_t)(dbx_plugin_session_t session,
                                            dbx_plugin_reginfo_t *infop,
                                            size_t reginfo_size,
                                            unsigned int *indexp,
                                            unsigned int *countp)
```

Ниже перечислены параметры процедуры обратного вызова dbx **regions**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенный массив структур dbx_plugin_region_t или NULL

reginfo_size
Размер отдельной структуры dbx_plugin_reginfo_t

indexp
Начальная или следующая область (нулевое значение соответствует первой области)

countp
Число областей

Ниже перечислены коды возврата процедуры обратного вызова dbx **regions**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *indexp* или *countp* равно NULL
- DBX_PLUGIN_BAD_ARG - недопустимое значение *reginfo_size* или **countp* == 0
- DBX_PLUGIN_UNAVAILABLE - сеанс представляет текущий процесс, доступ к областям памяти запрещен

нити

Процедура обратного вызова dbx **threads** позволяет получить информацию о нитях ядра процесса.

Существует два способа получения этой информации:

- Последовательно вызывая эту процедуру, получить информацию о каждой нити. Или
- Вызвать ее один раз для получения общего числа нитей и еще раз для получения информации сразу о всех нитях.

Если встраиваемый модуль передает ненулевой буфер *infop*, то программа **dbx** загружает в него записи, число которых указано в параметре **countp*, начиная с нити, указанной в параметре **indexp*.

Если встраиваемый модуль в параметре **countp* передает число, превышающее число оставшихся записей или равное ему, то программа **dbx** загружает все оставшиеся записи и обновляет параметр *countp* в соответствии с фактическим числом записей.

В случае загрузки последней записи, если *countp* меньше переданного значения, в параметре *indexp* указывается значение -1. В противном случае, значение *indexp* обновляется в соответствии с ИД нити для следующего запроса.

Примечание: Если переданное значение *countp* равно числу доступных записей, значение *countp* не изменяется, а в параметре *indexp* не указывается значение -1.

Если встраиваемый модуль передает нулевой буфер *infop*, то значения *indexp* и *countp* обновляются, как если бы *infop* не был нулевым.

```
typedef int (*dbx_plugin_threads_service_t)(dbx_plugin_session_t session,
                                            dbx _plugin_thrdinfo_t *infop,
                                            size_t thrdinfo_size,
                                            tid64_t *indexp,
                                            unsigned int *countp)
```

Ниже перечислены параметры процедуры обратного вызова dbx **threads**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенный массив структур dbx_plugin_thrdinfo_t или NULL

thrdinfo_size
Размер отдельной структуры dbx_plugin_thrdinfo_t

indexp
Идентификатор начальной или следующей нити (нулевое значение соответствует первой нити)

countp
Число нитей

Ниже перечислены коды возврата процедуры обратного вызова dbx **threads**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *indexp* или countp равно NULL
- DBX_PLUGIN_BAD_ID - недопустимое значение идентификатора **indexp*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *thrdinfo_size* или *countp ==0
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо информация отсутствует в дампе

pthreads

Процедура обратного вызова dbx **pthreads** позволяет получить информацию о нитях pthreads процесса, включая связи с нитями ядра.

Существует два способа получения этой информации:

- Последовательно вызывая эту процедуру, получить информацию о каждой нити pthread. Или
- Вызвать ее один раз для получения общего числа нитей pthread и еще раз для получения информации сразу о всех нитях pthread.

Если встраиваемый модуль передает ненулевой буфер *infop*, то программа **dbx** загружает в него записи, число которых указано в параметре **countp*, начиная с нити pthread, указанной в параметре **indexp*.

Если встраиваемый модуль в параметре **countp* передает число, превышающее число оставшихся записей, то программа **dbx** загружает все оставшиеся записи. Программа **dbx** обновляет значение *countp* в соответствии с фактическим числом записанных записей, а *indexp* в соответствии с описателем нити pthread для следующего запроса.

В случае загрузки последней записи в параметре *indexp* указывается значение -1. Если встраиваемый модуль передает нулевой буфер *infop*, то значения *indexp* и *countp* все равно обновляются, как если бы *infop* не был нулевым.

Если запрос первой нити pthread приводит к обновлению параметра *countp* нулевым значением, то данный процесс не поддерживает нити pthread.

```
typedef int (*dbx_plugin_pthreads_service_t)(dbx_plugin_session_t session,
                                             dbx_plugin_pthinfo_t *infop,
                                             size_t pthrdinfo_size,
                                             pthdb_pthread_t *indexp,
                                             unsigned int *countp)
```

Ниже перечислены параметры процедуры обратного вызова dbx **pthreads**:

Параметр
Описание

session
Идентификатор сеанса

infop
Выделенный массив структур dbx_plugin_pthinfo_t или NULL

pthrdinfo_size
Размер отдельной структуры dbx_plugin_pthrdinfo_t

indexp
Описатель начальной или следующей нити pthread (нулевое значение соответствует первой нити pthread, значение DBX_PLUGIN_PTHREAD_CURRENT соответствует текущей нити pthread в **dbx**)

countp
Число нитей pthread

Ниже перечислены коды возврата процедуры обратного вызова dbx **pthreads**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *indexp* или *countp* равно NULL
- DBX_PLUGIN_BAD_ARG - недопустимое значение *pthrdinfo_size* или **countp* == 0

get_thread_context

Процедура обратного вызова dbx **get_thread_context** позволяет прочитать данные из регистра общего назначения, специального регистра и регистра чисел с плавающей точкой. Программа **dbx** задает параметр *contextp*.

```
typedef int (*dbx_plugin_reg_service_t)(dbx_plugin_session_t session,
                                        uint64_t reg_flags,
                                        uint64_t id,
                                        dbx_plugin_context_t *contextp,
                                        size_t context_size)
```

Ниже перечислены параметры процедуры обратного вызова dbx **get_thread_context**:

Параметр
Описание

session
Идентификатор сеанса

reg_flags
Логическое объединение по крайней мере одного из следующих флагов: DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS, DBX_PLUGIN_REG_EXT

id tid нити ядра (tid64_t)

contextp
Выделенная структура dbx_plugin_context_t

context_size
Размер структуры dbx_plugin_context_t. Размер структуры dbx_plugin_extctx_t применяется совместно с флагом DBX_PLUGIN_REG_EXT. Структура dbx_plugin_extctx_t представляет собой расширенную версию структуры dbx_plugin_context_t.

Ниже перечислены коды возврата процедуры обратного вызова dbx **get_thread_context**:

- DBX_PLUGIN_SUCCESS.
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*.
- DBX_PLUGIN_BAD_ID - недопустимое значение *id*.
- DBX_PLUGIN_BAD_ARG - недопустимое значение *reg_flags* или *context_size*.
- DBX_PLUGIN_BAD_POINTER - значение *contextp* равно NULL
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо нить работает в режиме ядра, в котором доступ к регистрам запрещен.

set_thread_context

Процедура обратного вызова dbx **set_thread_context** позволяет записать данные в регистр общего назначения, специальный регистр и регистр чисел с плавающей точкой.

```
typedef int (*dbx_plugin_reg_service_t) (dbx_plugin_session_t session,
                    uint64_t reg_flags,
                                uint64_t id,
                                dbx_plugin_context_t *contextp,
                                size_t context_size)
```

Ниже перечислены параметры процедуры обратного вызова dbx **set_thread_context**:

Параметр
Описание

session
Идентификатор сеанса

reg_flags
Логическое объединение по крайней мере одного из следующих флагов: DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS, DBX_PLUGIN_REG_EXT

id tid нити ядра (tid64_t)

contextp
Выделенная структура dbx_plugin_context_t

context_size
Размер структуры dbx_plugin_context_t. Размер структуры dbx_plugin_extctx_t применяется совместно с флагом DBX_PLUGIN_REG_EXT. Структура dbx_plugin_extctx_t представляет собой расширенную версию структуры dbx_plugin_context_t.

Ниже перечислены коды возврата процедуры обратного вызова dbx **set_thread_context**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ID - недопустимое значение *id*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *reg_flags* или *context_size*
- DBX_PLUGIN_BAD_POINTER - значение *contextp* равно NULL
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо нить работает в режиме ядра, в котором доступ к регистрам запрещен

get_pthread_context

Процедура обратного вызова dbx **get_pthread_context** предназначена для чтения данных из регистров общего назначения, регистров для чисел с плавающей точкой и специальных регистров нити pthread. Программа **dbx** задает параметр *contextp*.

```
typedef int (*dbx_plugin_reg_service_t)(dbx_plugin_session_t session,
                                        uint64_t reg_flags,
                                        uint64_t id,
                                        dbx_plugin_context_t *contextp,
                                        size_t context_size)
```

Ниже перечислены параметры процедуры обратного вызова dbx **get_pthread_context**:

Параметр
Описание

session
Идентификатор сеанса

reg_flags
Логическое объединение по крайней мере одного из следующих флагов: DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS, DBX_PLUGIN_REG_EXT

id Описатель нити pthread (pthdb_pthread_t)

contextp
Выделенная структура dbx_plugin_context_t

context_size
Размер структуры dbx_plugin_context_t. Размер структуры dbx_plugin_extctx_t применяется совместно с флагом DBX_PLUGIN_REG_EXT. Структура dbx_plugin_extctx_t представляет собой расширенную версию структуры dbx_plugin_context_t.

Ниже перечислены коды возврата процедуры обратного вызова dbx **get_pthread_context**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ID - недопустимое значение *id*.
- DBX_PLUGIN_BAD_ARG - недопустимое значение *reg_flags* или *context_size*
- DBX_PLUGIN_BAD_POINTER - значение *contextp* равно NULL
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо нить работает в режиме ядра, в котором доступ к регистрам запрещен

set_pthread_context

Процедура обратного вызова dbx **set_pthread_context** позволяет записать данные в регистр общего назначения, специальный регистр и регистр чисел с плавающей точкой нити pthread.

```
typedef int (*dbx_plugin_reg_service_t)(dbx_plugin_session_t session,
                                        uint64_t reg_flags,
                                        uint64_t id,
                                        dbx_plugin_context_t *contextp,
                                        size_t context_size)
```

Ниже перечислены параметры процедуры обратного вызова dbx **set_pthread_context**:

Параметр
Описание

session
Идентификатор сеанса

reg_flags
Логическое объединение по крайней мере одного из следующих флагов: DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS, DBX_PLUGIN_REG_EXT

id Описатель нити Pthread (pthdb_pthread_t)

contextp
Выделенная структура dbx_plugin_context_t

context_size
Размер структуры dbx_plugin_context_t. Размер структуры dbx_plugin_extctx_t применяется совместно с флагом DBX_PLUGIN_REG_EXT. Структура dbx_plugin_extctx_t представляет собой расширенную версию структуры dbx_plugin_context_t.

Ниже перечислены коды возврата процедуры обратного вызова dbx **set_pthread_context**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ID - недопустимое значение *id*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *reg_flags* или *context_size*
- DBX_PLUGIN_BAD_POINTER - значение *contextp* равно NULL
- DBX_PLUGIN_UNAVAILABLE - процесс неактивен, либо нить ядра, связанная с нитью pthread, работает в режиме ядра, в котором доступ к регистрам запрещен

read_memory

Процедура обратного вызова dbx **read_memory** позволяет прочитать данные из адресного пространства процесса. Программа **dbx** задает параметр buffer.

```
typedef int (*dbx_plugin_mem_service_t)(dbx_plugin_session_t session,
                                        uint64_t addr,
                                        void *buffer,
                                        size_t len)
```

Ниже перечислены параметры процедуры обратного вызова **read_memory**:

Параметр
Описание

session
Идентификатор сеанса

адрес
Начальный адрес для чтения

buffer
Выделенный буфер для сохранения сохранения содержимого памяти

len
Число байт для чтения

Ниже перечислены коды возврата процедуры обратного вызова dbx **read_memory**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *buffer* равно NULL
- DBX_PLUGIN_UNAVAILABLE - не удалось выполнить чтение данных из *addr*

write_memory

Процедура обратного вызова dbx **write_memory** позволяет записать данные в адресное пространство процесса.

```
typedef int (*dbx_plugin_mem_service_t)(dbx_plugin_session_t session,
                                        uint64_t addr,
                                        void *buffer,
                                        size_t len)
```

Ниже перечислены параметры процедуры обратного вызова **write_memory**:

Параметр
Описание

session
Идентификатор сеанса

адрес
Начальный адрес для записи

buffer
Выделенный и инициализированный буфер

len
Число байт для записи

Ниже перечислены коды возврата процедуры обратного вызова dbx **write_memory**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_POINTER - значение *buffer* равно NULL
- DBX_PLUGIN_UNAVAILABLE - не удалось записать данные в *addr*

locate_symbol

Процедура обратного вызова dbx **locate_symbol** позволяет преобразовать символьные имена в адреса.

Встраиваемый модуль должен инициализировать поля *name* и *mod* каждой записи массива символьных параметров. Поле *name* задает имя символа, который необходимо найти. Поле *mod* задает индекс модуля для поиска. Если указать в поле *mod* значение -1, то поиск будет выполнен во всех модулях.

Программа **dbx** заполняет поле *addr*. Для всех неизвестных символов указывается нулевой адрес. Если символ найден в результате поиска во всех модулях, **dbx** указывает в поле *mod* фактический индекс модуля этого символа.

```
typedef int (*dbx_plugin_sym_service_t)(dbx_plugin_session_t session,
                                        dbx_plugin_sym_t *symbols,
                                        size_t syminfo_size,
                                        unsigned int count)
```

Ниже перечислены параметры процедуры обратного вызова **locate_symbol**:

Параметр
Описание

session
Идентификатор сеанса

symbols
Выделенный массив структур dbx_plugin_sym_t structures, для которого инициализированы поля *name* и *mod*

syminfo_size
Размер структуры dbx_plugin_sym_t

count
Число символов для преобразования

Ниже перечислены коды возврата процедуры обратного вызова dbx **locate_symbol**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *syminfo_size*
- DBX_PLUGIN_BAD_POINTER - значение *symbols* равно NULL

what_function

Процедура обратного вызова dbx **what_function** позволяет преобразовать текстовые адреса в имена.

Встраиваемый модуль должен инициализировать поле *addr* каждой записи массива символьных параметров. Поле *addr* задает адрес инструкции, которую необходимо идентифицировать, в функции.

Программа **dbx** заполняет поле *name*. Для всех неизвестных текстовых адресов задается нулевое имя. Программа **dbx** указывает в поле *mod* фактический индекс модуля текстового адреса.

```
typedef int (*dbx_plugin_sym_service_t)(dbx_plugin_session_t session,
                                        dbx_plugin_sym_t *symbols,
                                        size_t syminfo_size,
                                        unsigned int count)
```

Ниже перечислены параметры процедуры обратного вызова **what_function**:

Параметр
Описание

session
Идентификатор сеанса

symbols
Выделенный массив структур dbx_plugin_sym_t, в которых в полях addr указаны текстовые адреса

syminfo_size
Размер структуры dbx_plugin_sym_t

count
Число адресов для преобразования

Ниже перечислены коды возврата процедуры обратного вызова dbx **what_function**:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *syminfo_size*
- DBX_PLUGIN_BAD_POINTER - значение *symbols* равно NULL

print

Процедура обратного вызова dbx **print** позволяет просмотреть информационный вывод и сообщения об ошибках.

```
typedef int (*dbx_plugin_print_service_t)(dbx_plugin_session_t session,
                                          int print_mode,
                                          char *message)
```

Ниже перечислены параметры процедуры обратного вызова dbx **print**:

Параметр
Описание

session
Идентификатор сеанса

print_mode
DBX_PLUGIN_PRINT_MODE_OUT или DBX_PLUGIN_PRINT_MODE_ERR

message
Строка символов для отображения в программе **dbx**

Ниже перечислены коды возврата процедуры обратного вызова dbx **print**:
- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *print_mode*
- DBX_PLUGIN_BAD_POINTER - значение *message* равно NULL

alias

Процедура обратного вызова dbx **alias** позволяет создать псевдоним для команды.

В соответствии с синтаксисом команды **plugin dbx** в каждом вызове команды встраиваемого модуля пользователь dbx должен указать префикс `Имя модуля`. Для того чтобы сократить длину таких вызовов, программа **dbx** позволяет создавать для встраиваемых модулей псевдонимы.

Псевдоним описывается с помощью параметров *alias* и *expansion*. При этом применяется такой же синтаксис, как в команде alias программы **dbx**.

Ниже приведен пример вызовов процедуры обратного вызова dbx **alias**:
```
alias("intprt", "plugin xyz interpret");
alias("intprt2(addr, count, format)", "addr / count format; plugin xyz interpret addr");
```

Примечание: Если в запросе на создание псевдонима указать имя уже существующего псевдонима, запрос будет отклонен и будет выдано предупреждающее сообщение. Разработчикам встраиваемых модулей рекомендуется создавать псевдонимы таким образом, чтобы пользователи могли в дальнейшем самостоятельно устранять конфликты псевдонимов. Например, определения псевдонимов можно загружать из файла конфигурации, поставляемого вместе со встраиваемым модулем.
```
typedef int (*dbx_plugin_alias_service_t)(dbx_plugin_session_t session,
                                          const char *alias,
                                          const char *expansion)
```

Ниже перечислены параметры процедуры обратного вызова dbx **alias**:

Параметр
Описание

session
Идентификатор сеанса

alias
Строка символов, в которой указано имя псевдонима и дополнительные параметры

expansion
Строка символов, в которой указано расширение псевдонима

Ниже перечислены коды возврата процедуры обратного вызова dbx **alias**:
- DBX_PLUGIN_SUCCESS

- DBX_PLUGIN_BAD_SESSION - недопустимое значение *session*
- DBX_PLUGIN_BAD_ARG - недопустимое значение *alias*
- DBX_PLUGIN_BAD_POINTER - значение *alias* или expansion равно NULL
- DBX_PLUGIN_UNAVAILABLE - псевдоним с таким именем уже существует

Пример

1. Пример создания команд **help** и **hello**:

```
example.c:

#include <sys/dbx_plugin.h>

dbx_plugin_session_t sid;
dbx_plugin_services_t dbx;

static void usage(void);
static void hello_cmd(void);

int
dbx_plugin_version(void) {
    return DBX_PLUGIN_VERSION_1;
}

int dbx_plugin_session_init(dbx_plugin_session_t session,
                            const dbx_plugin_services_t *servicep) {
    /* записать идентификатор сеанса */
    sid= session;

    /* записать службу dbx */
    memcpy(&dbx, servicep, sizeof(dbx_plugin_services_t));

    (*(dbx.alias))(sid, "hello", "plugin example hello");
    (*(dbx.alias))(sid, "help", "plugin example help");

    return 0;

}

void
dbx_plugin_session_command(dbx_plugin_session_t session,
                           int argc,
                           char *const argv[]) {
    if (argc == 0 || (argc == 1 && strcmp(argv[0], "help") == 0)) {

 usage();
 return;
    }
    if (argc == 1 && strcmp(argv[0], "hello") == 0) {
 hello_cmd();
 return;
    }
    (*(dbx.print))(sid,DBX_PLUGIN_PRINT_MODE_ERR,
     "не удалось распознать команду\n");

}

void
dbx_plugin_session_event(dbx_plugin_session_t session,
                         int event,
                         dbx_plugin_event_info_t *event_infop) {
    /* игнорировать уведомления о событиях */
}

void
dbx_plugin_session_destroy(dbx_plugin_session_t session){
```

```
    /* не выполнять очистку */
}

static
void
usage(void) {
    (*(dbx.print))(sid,DBX_PLUGIN_PRINT_MODE_OUT,
                       "Команды для встраиваемого модуля \"example\":\n\n" \
                       "   help  - отображается эта справочная информация\n" \
                       "   hello - отображается приветствие\n" \
     "\n");
}

static
void
hello_cmd(void) {
    (*(dbx.print))(sid,DBX_PLUGIN_PRINT_MODE_OUT,
     "Здравствуй, мир dbx!\n");

}
```

example.exp:

```
dbx_plugin_version
dbx_plugin_session_init
dbx_plugin_session_command
dbx_plugin_session_event
dbx_plugin_session_destroy
```

2. Команда компиляции примера встраиваемого модуля:

```
cc -q64 -o libdbx_example.so example.c -bM:Sre -bE:example.exp -bnoentry
```

Список команд dbx

Полный список команд программы отладки **dbx** приведен в книге *Справочник по командам*.

Команды программы **dbx** позволяют выполнять следующие задачи:

Установка и удаление точек прерывания

Команда	Описание
clear	Удаляет все точки прерывания в заданной строке исходного файла.
cleari	Удаляет все точки прерывания с заданным адресом.
delete	Удаляет точки трассировки и прерывания с указанными номерами.
status	Показывает активные на данный момент команды **trace** и **stop**.
stop	Прекращает выполнение прикладной программы.

Выполнение программ

Команда	Описание
cont	Возобновляет выполнение программы с текущей точки прерывания до конца программы или до следующей точки прерывания.
Отключить	Выход из программы отладки без завершения работы приложения.
down	Передвигает функцию вниз по стеку.
goto	Передает управление на указанную строку исходного кода программы.
gotoi	Изменяет адреса счетчика команд.
next	Приложение выполняется до следующей строки исходного кода.
nexti	Приложение выполняется до следующей команды исходного кода.
rerun	Запускает приложение.
return	Продолжает выполнять прикладную программу, пока не будет достигнут оператор возврата в указанную процедуру.
run	Запускает приложение.
skip	Возобновляет работу приложения с текущей точки прерывания.
step	Выполняет одну строку исходного кода.

Команда	Описание
stepi	Выполняет одну команду исходного кода.
up	Передвигает функцию вверх по стеку.

Трассировка программ

Команда	Описание
трассировка	Печатает данные трассировки.
tracei	Включает трассировку.
where	Выдает список всех активных процедур и функций.

Завершение работы программы

Команда	Описание
quit	Выход из программы отладки **dbx**.

Просмотр исходного кода

Команда	Описание
edit	Открывает окно редактора с указанным файлом.
file	Переход от текущего файла с исходным кодом к указанному файлу.
функция	Изменяет текущую функцию на указанную функцию или процедуру.
list	Выдает текст текущего файла с исходным кодом.
listi	Выдает список команд из исходного кода приложения.
перемещение	Переход на указанную строку текста.
/ (искомая-строка)	Поиск по образцу в текущем файле с исходным кодом (в направлении вперед)
? (Поиск)	Поиск по образцу в текущем файле с исходным кодом (в направлении назад)
use	Задает список каталогов для поиска файла.

Печать и изменение переменных, выражений и типов

Команда	Описание
assign	Присваивает значение переменной.
case	Изменяет способ интерпретации символов программой **dbx**.
dump	Выводит имена и значения переменных указанной процедуры.
print	Печатает значение выражения, либо выполняет процедуру и печатает код возврата.
set	Присваивает значение переменной среды.
unset	Аннулирует присвоенное ранее значение переменной среды.
whatis	Выводит описание компонентов прикладной программы.
whereis	Выводит полные имена всех символов, совпадающих с указанным идентификатором.
which	Выводит полное имя указанного идентификатора.

Отладка нити

Команда	Описание
attribute	Выдает информацию о заданном атрибуте или о всех атрибутах.
condition	Выводит информацию об указанной или о всех условных переменных.
mutex	Выводит информацию об указанной или о всех взаимных блокировках.
thread	Отслеживает работу нити и выводит собранную информацию.
tstophwp	Устанавливает остановку в аппаратной точке наблюдения уровня нитей.
ttracehwp	Устанавливает трассировку в аппаратной точке наблюдения уровня нитей.
tstop	Устанавливает для нити остановку в точке прерывания уровня исходного кода.
tstopi	Устанавливает для нити остановку в точке прерывания уровня инструкций.
ttrace	Устанавливает для нити трассировку на уровне исходного кода.
ttracei	Устанавливает для нити трассировку на уровне инструкций.
tnext	Передает выполнение нити к следующей строке исходного кода.
tnexti	Передает выполнение нити к следующей машинной инструкции.
tstep	Выполняет одну строку исходного кода нити.

Команда	Описание
tstepi	Выполняет одну машинную инструкцию нити.
tskip	Пропускает точки прерывания для нити.

Отладка в параллельном режиме

Команда	Описание
multproc	Включает или выключает параллельный режим отладки.

Вызов процедуры

Команда	Описание
call	Выполняет объектный код, связанный с заданной процедурой или функцией.
print	Печатает значение выражения, либо выполняет процедуру и печатает код возврата.

Обработка сигналов

Команда	Описание
catch	Включение режима перехвата сигнала перед его отправкой приложению.
ignore	Выключение режима перехвата сигнала перед его отправкой приложению.

Отладка на машинном уровне

Команда	Описание
display memory	Выводит содержимое памяти.
gotoi	Изменяет адреса счетчика команд.
map	Выдает таблицу адресов и информацию загрузчика о прикладной программе.
nexti	Выполняет прикладную программу до следующей машинной команды.
registers	Показывает значения регистров общего назначения, системных регистров, регистров с плавающей точкой и регистра команд.
stepi	Выполняет одну команду исходного кода.
stopi	Устанавливает точку прерывания в указанной позиции.
tracei	Включает трассировку.

Управление средой отладки

Команда	Описание
alias	Присваивает псевдонимы командам **dbx** и выводит информацию о них.
help	Выдает справочную информацию по командам программы **dbx** и другим вопросам.
prompt	Изменяет стандартное приглашение **dbx** на указанную строку.
экран	Открывает окно Xwindow, в которое будет отправляться вывод команды **dbx**.
sh	Передает оболочке команду для выполнения.
source	Считывает команды **dbx** из файла.
unalias	Удаляет псевдоним.

Обзор средства ведения протокола ошибок

Процесс регистрации ошибки начинается с момента, когда ошибка обнаружена модулем операционной системы.

Сегмент кода, отвечающий за обнаружение ошибок, передает сведения об ошибке либо в службы ядра **errsave** и **errlast**, либо в функцию **errlog**. Затем информация об ошибке передается в специальный файл **/dev/error**. Вместе с данными об ошибке записывается время ее обнаружения. Демон **errdemon** постоянно проверяет наличие новых записей в файле **/dev/error**, а при поступлении новых данных выполняет стандартную процедуру обработки.

Прежде чем добавить запись в протокол ошибок, демон **errdemon** сравнивает метку, полученную от ядра или приложения, с содержимым реестра шаблонов ошибок. Если в реестре есть запись, соответствующая метке, демон начинает сбор данных из других областей системы.

Для создания записи в протоколе ошибок демон **errdemon** считывает шаблон из реестра, имя ресурса блока, обнаружившего ошибку, и сведения об ошибке. Если ошибка свидетельствует об аппаратной неполадке и для нее предусмотрены специальные данные в реестре аппаратного обеспечения (VPD), то демон считывает VPD из ODM. При обращении к протоколу ошибок с помощью SMIT или команды **errpt** данные протокола форматируются в соответствии с шаблонами в реестре шаблонов и представляются в виде краткого или подробного отчета. Записи можно также получить с помощью функций **liberrlog**, **errlog_open**, **errlog_close**, **errlog_find_first** , **errlog_find_next**, **errlog_find_sequence** , **errlog_set_direction** и **errlog_write**. **errlog_write** обеспечивает ограниченную возможность обновления.

Большинство записей в протоколе ошибок связано с программными и аппаратными неполадками, однако в нем могут быть и информационные сообщения.

Команда **diag** применяется для диагностики аппаратных неполадок на основе содержимого протокола ошибок. Для правильной диагностики новых неполадок система удаляет из протокола записи об аппаратных ошибках старше 90 дней. Записи о программных ошибках удаляются через 30 дней после занесения в протокол.

Рекомендуем вам ознакомиться со следующими терминами:

Термин	Описание
ИД ошибки	32-разрядный шестнадцатеричный код CRC, однозначно идентифицирующий конкретную ошибку. Всем шаблонам ошибок в реестре присвоены уникальные идентификаторы.
метка ошибки	Символьная строка, назначенная идентификатору ошибки.
протокол ошибок	Файл, в котором хранятся сведения об ошибках и сбоях, обнаруженных системой.
запись протокола ошибок	Запись протокола ошибок, в которой описан аппаратный или программный сбой, или сообщение оператора. В записи хранятся все данные об ошибке.
шаблон ошибки	Формальное описание сведений, которые будут показаны об определенной ошибке в отчете, включая тип и класс ошибки, ее возможные причины и рекомендуемые действия. Множество шаблонов образует реестр шаблонов ошибок.

Информация, связанная с данной:

Специальные файлы протоколов ошибок

Команда errsave

Команда errlog

Команда crontab

Команда errclear

Команда errdead

Команда errdemon

Команда errinstall

Команда errlogger

Команда errmsg

Команда errpt

Команда errstop

Команда errupdate

Команда odmadd

Команда errstop

Команда odmget

Команда snap

Средство ведения протокола ошибок

Средство ведения протокола ошибок заносит в протокол ошибок сообщения о сбоях программного и аппаратного обеспечения.

Средству ведения протокола ошибок посвящены следующие разделы:

В AIX версии 4 некоторые команды по работе с протоколом ошибок поставляются в дополнительном пакете **bos.sysmgt.serv_aid**. Базовая система (**bos.rte**) содержит следующие службы записи сообщений об ошибках в файл протокола ошибок:

- процедуры errlog
- службы ядра **errsave** и **errlast**
- драйвер устройства ошибок (**/dev/error**)
- демон **error**
- команда **errstop**

Команды, необходимые для установки лицензионных программ (**errinstall** and **errupdate**), также включены в пакет **bos.rte**. Инструкции по переносу файла системного протокола ошибок в систему, в которой установлен пакет Software Service Aids, приведены в разделе Передача протокола ошибок в другую систему.

Работа с протоколом ошибок

Функция ведения протокола ошибок автоматически включается во время инициализации системы с помощью сценария **rc.boot** и автоматически выключается во время завершения работы системы с помощью сценария **shutdown**.

Команда **diag** анализирует записи протокола об аппаратных ошибках. По умолчанию записи об аппаратных ошибках хранятся в протоколе ошибок в течение 90 дней. Если из протокола будут удалены некоторые записи об аппаратных ошибках, занесенные в течение последних 90 дней, то эффективность анализа протокола понизится.

Передача протокола ошибок в другую систему

Команды **errclear**, **errdead**, **errlogger**, **errmsg** и **errpt** входят в состав дополнительного пакета Software Service Aids (**bos.sysmgt.serv_aid**). Этот пакет применяется для создания отчетов на основе протокола ошибок и удаления записей из протокола ошибок. Вы можете установить пакет Software Service Aids в своей системе, либо передать файл с протоколом ошибок в другую систему, содержащую этот пакет.

С помощью следующей команды узнайте, в каком каталоге расположен файл с системным протоколом ошибок:

```
/usr/lib/errdemon -l
```

Этот файл можно передать в другую систему несколькими способами. Они перечислены ниже:

- Скопируйте файл в смонтированную удаленную файловую систему с помощью команды **cp**
- Скопируйте файл по сетевому соединению с помощью команды **rcp**, **ftp** или **tftp**
- Скопируйте файл на съемный носитель с помощью команды **tar** или **backup**, а затем восстановите его в другой системе.

Для создания отформатированных отчетов на основе протокола ошибок, скопированного из другой системы, служит команда **errpt** с флагом **-i**. С флагом **-i** можно задать каталог, в котором расположен файл протокола ошибок, если этот файл расположен не в каталоге по умолчанию. Для удаления записей из протокола ошибок, скопированного из другой системы, служит команда **errclear** с флагом **-i**.

Параметры протокола ошибок

Вы можете изменить имя и расположение файла протокола ошибок, а также размер внутреннего буфера ошибок. Кроме того, можно задать опцию занесения в протокол информации о повторных ошибках.

Просмотр текущих значений параметров

Для просмотра текущих значений параметров вызовите команду **/usr/lib/errdemon -l**. В выводе этой команды указывается имя файла протокола ошибок, размер файла протокола ошибок и размер буфера, которые в настоящий момент заданы в базе данных конфигурации протокола ошибок.

Изменение расположения файла протокола

Для того чтобы изменить имя файла протокола ошибок, вызовите команду **/usr/lib/errdemon -i** *имя-файла*. Указанное имя файла будет сохранено в базе данных конфигурации протокола ошибок. После этого будет автоматически перезапущен демон ведения протокола ошибок.

Изменение размера файла протокола

Для того чтобы изменить максимальный размер файла протокола ошибок, введите:

```
/usr/lib/errdemon -s размер-протокола
```

Указанный максимальный размер файла протокола будет сохранен в базе данных конфигурации протокола ошибок. После этого будет автоматически перезапущен демон ведения протокола ошибок. Если установленное ограничение меньше текущего размера файла протокола, то текущий файл переименовывается путем добавления суффикса **.old** , после чего создается новый файл с указанным ограничением. Заданный объем памяти резервируется для файла протокола ошибок и недоступен для других файлов. В связи с этим не следует устанавливать очень большое ограничение. Однако если размер протокола будет очень маленьким, то из него преждевременно может быть удалена важная информация. После того как размер файла протокола достигает указанного ограничения, *начинается новый цикл записи*, то есть самые старые записи заменяются на новые.

Изменение размера буфера

Для того чтобы изменить размер внутреннего буфера драйвера устройства протокола ошибок, введите:

```
/usr/lib/errdemon -B размер-буфера
```

Указанный размер буфера сохраняется в базе данных конфигурации протокола ошибок. При увеличении размера буфера внутренний буфер немедленно расширяется. Если новое значение меньше текущего размера буфера, то оно вступает в силу при первом запуске демона ведения протокола ошибок после перезапуска системы. Размер буфера не должен быть меньше значения по умолчанию, составляющего 8 Кб. Указанный размер округляется с избытком до значения, кратного размеру страницы памяти (4 Кб). Память, выделенная внутреннему буферу драйвера устройства протокола ошибок, недоступна другим процессам (другими словами, буфер размещается в закрепленной памяти).

Если буферу будет выделен слишком большой объем памяти, то производительность системы может снизиться. Однако если размер буфера будет слишком маленьким, то буфер может переполниться. Это может произойти в том случае, если записи об ошибках записываются в буфер чаще, чем считываются из буфера в файл протокола. После заполнения буфера все новые записи отбрасываются до тех пор, пока в буфере не освободиться достаточный объем памяти. В этом случае создается запись протокола ошибок с информацией о неполадке. При обнаружении такой записи можно увеличить размер буфера.

Изменение способа обработки повторных ошибок

В AIX 5.1 и выше демон ведения протокола ошибок по умолчанию удаляет записи о повторных ошибках, просматривая все зарегистрированные ошибки. Ошибка считается повторной, если она произошла в течение

заданного интервала времени после возникновения аналогичной ошибки. Этот интервал времени можно изменить с помощью команды **/usr/lib/errdemon -t интервал-времени**. Значение по умолчанию составляет 10000 (10 секунд). Это значение указывается в мс.

С помощью флага **-m максимальное-число-повторов** можно указать, сколько повторных ошибок должно произойти прежде, чем в протокол ошибок будет занесена повторная запись об ошибке. Значение по умолчанию равно 1000. Оно означает, что при возникновении 1001 повторной ошибки в протокол будет занесена еще одна запись об ошибке, независимо от того, истек ли указанный интервал времени.

Если, например, программа обработки ошибок устройства часто создает записи об одинаковых ошибках, то большинство этих записей не будут занесены в протокол. В протокол будет занесена запись о первой ошибке. Информация о последующих идентичных ошибках не будет заноситься в протокол, но ошибки будут подсчитываться. По истечении указанного интервала времени, при достижении ограничения **максимальное-число-повторов**, либо при занесении записи о другой ошибке в протокол будет добавлена еще одна запись о повторяющейся ошибке, в которой будет указано время возникновения первой и последней ошибки и число повторных ошибок.

Примечание: Интервал времени отсчитывается с момента возникновения последней ошибки, а не с момента возникновения первой ошибки, т.е. его значение обнуляется при занесении в протокол новой записи об ошибке. Обратите внимание, что повторной считается ошибка, которая в точности совпадает с одной из предыдущих ошибок. Например, если подробная информация об ошибке немного отличается от аналогичной информации об одной из предыдущих ошибок, то такая ошибка считается уникальной, и для нее создается отдельная запись протокола.

Удаление записей из протокола ошибок

Записи удаляются из протокола ошибок при вызове команды **errclear** пользователем root, при вызове команды **errclear** ежедневно выполняемым заданием **cron**, либо при начале нового цикла записи в файл протокола ошибок после того, как был достигнут максимальный размер файла. После того как размер файла протокола ошибок достигает ограничения, указанного в базе данных конфигурации протокола ошибок, самые старые записи протокола начинают заменяться на новые записи.

Автоматическое удаление

По умолчанию в файле **crontab** содержатся команды, автоматически удаляющие записи об аппаратных ошибках, занесенные более 90 дней назад, и остальные записи, занесенные более 30 дней назад. Для просмотра записей файла **crontab** локальной системы введите:

```
crontab -l команда
```

Для изменения этих записей введите:

```
crontab -e команда
```

errclear, команда

Команда **errclear** позволяет выборочно удалить записи из протокола ошибок. В качестве критерия выбора записей можно указать ИД ошибки, порядковый номер, метку ошибки, имя ресурса, класс ресурса, класс ошибки и тип ошибки. Кроме того, необходимо указать срок хранения записей в протоколе. Команда удалит все записи, которые соответствуют заданному критерию и хранятся в протоколе дольше указанного времени.

Включение и выключение функции ведения протокола для события

Вы можете выключить функцию ведения протокола для отдельного события путем изменения поля **Log** или **Report** в шаблоне записи об ошибке, связанном с данным событием. Для изменения параметров события служит команда **errupdate**.

Просмотр списка событий, для которых не ведется протокол

Для того чтобы просмотреть список событий, для которых не ведется протокол, введите:

```
errpt -t -F Log=0
```

Если для события не ведется протокол, то информация об этом событии не заносится в файл протокола ошибок.

Просмотр событий, для которых не создается отчет

Для того чтобы просмотреть список событий, для которых не создается отчет, введите:

```
errpt -t -F Report=0
```

Информация о событиях, для которых не создается отчет, заносится в файл протокола ошибок, однако не выводится командой **errpt**.

Изменение текущих параметров события

Текущие параметры события можно изменить с помощью команды **errupdate**. Эта команда считывает входные данные из файла или из стандартного ввода.

В приведенном ниже примере применяется стандартный ввод. Для того чтобы выключить функцию создания отчета для события **ERRLOG_OFF** (номер ошибки - 192AC071), запустите команду **errupdate**, введя следующую информацию:

```
errupdate <Enter>
=192AC071: <Enter>
Report=False <Enter>
<Ctrl-D>
<Ctrl-D>
```

Занесение в протокол информации об обслуживании

С помощью команды **errlogger** системный администратор может добавлять записи в протокол ошибок. При выполнении обслуживания системы рекомендуется заносить в системный протокол ошибок информацию о выполненных действиях, например, об очистке протокола ошибок, замене аппаратного компонента или применении исправления.

Команда **ras_logger** позволяет занести в протокол информацию о любой ошибке. Она предназначена для тестирования новых шаблонов ошибок и занесения сообщений об ошибках в протокол из сценария оболочки.

Копирование сообщений из системного протокола в протокол ошибок

Некоторые приложения заносят информацию об ошибках и других событиях в системный протокол. Для просмотра записей об ошибках и записей системного протокола в одном отчете перенаправьте сообщения системного протокола в протокол ошибок. Это можно сделать, указав в файле конфигурации системного протокола (**/etc/syslog.conf**) целевой объект *errlog*. За дополнительной информацией обратитесь к описанию демона **syslogd**.

Копирование записей протокола ошибок в системный протокол

Записи об ошибках могут заноситься в файл **syslog** командой **logger**. Одновременно эти записи будут заноситься в протокол ошибок. Например, для того чтобы в этот файл заносились сообщения системы, добавьте объект errnotify со следующим описанием:

```
errnotify:
        en_name = "syslog1"
        en_persistenceflg = 1
        en_method = "logger Msg from Error Log: `errpt -l $1 | grep -v 'ERROR_ID TIMESTAMP'`"
```

Например, создайте файл **/tmp/syslog.add**, содержащий указанный выше текст. Затем вызовите команду **odmadd /tmp/syslog.add** (эту команду разрешено выполнять только пользователю root).

Дополнительная информация об одновременном уведомлении об ошибках приведена в разделе Уведомление об ошибках.

Извещение об ошибках

Объектный класс Извещение об ошибках определяет условия и действия, выполняемые при записи сообщений об ошибках в протокол ошибок. Эти условия и действия задаются пользователем в объекте извещения об ошибках.

Каждый раз, когда в протокол заносится ошибка, демон **извещения об ошибках** определяет, совпадает ли запись протокола ошибок с критериями выбора какого-либо объекта класса извещения об ошибках. Для каждого объекта, для которого обнаружено совпадение, демон выполняет запрограммированное действие, называемое также *способом извещения*.

Объектный класс Извещение об ошибках располагается в файле **/etc/objrepos/errnotify**. Объекты извещения об ошибках добавляются к объектному классу с помощью команд Администратора объектных данных (ODM). Добавлять объекты в класс извещений об ошибках разрешено только процессам, которые запускаются пользователем root. Объекты извещения об ошибках содержат следующие дескрипторы:

en_alertflg
Указывает, следует ли предупреждать об ошибке. Этот дескриптор используется агентами предупреждений, связанными с программами сетевого управления, с помощью архитектуры извещения SNA. Допустимы следующие значения дескриптора:

TRUE извещение разрешено

FALSE
извещение запрещено

en_class
Идентифицирует класс записей протокола ошибок, проверяемых на совпадение. Допустимы следующие значения дескриптора **en_class**:

H Класс аппаратных ошибок

S Класс программных ошибок

O Сообщения команды **errlogger**

U Неопределенная ошибка

en_crcid
Указывает идентификатор ошибки. Идентификатор ошибки может быть любым числовым значением, допустимым в качестве значения атрибута класса объектов предопределенного атрибута (PdAt). Команда **errpt** показывает идентификаторы ошибок в шестнадцатеричном формате. Например, для выбора записи, которая в выводе команды **errpt** выглядит как `IDENTIFIER: 67581038`, укажите `en_crcid = 0x67581038`.

en_dup
Если установлен, указывает, следует ли проверять определенные в ядре повторные ошибки. Допустимы следующие значения дескриптора **en_dup**:

TRUE Повторная ошибка.

FALSE
Ошибка не повторная.

en_err64
Если установлен, указывает, следует ли проверять ошибки из 64-разрядной или из 32-разрядной среды. Допустимы следующие значения дескрипторов **en_err64**:

TRUE Ошибка в 64-разрядной среде.

FALSE
Ошибка в 32-разрядной среде.

en_label
Задает метку, связанную с конкретным идентификатором ошибки, согласно определению в выводе команды **errpt -t**.

en_method
Определяет запрограммированное пользователем действие (например, сценарий оболочки или строку команд), которое выполняется при занесении в протокол ошибки, совпадающей с критериями выбора данного объекта извещения об ошибках. Для выполнения этого действия демон извещения об ошибках применяет команду **sh -c**.

Ниже перечислены ключевые слова, которые автоматически развертываются демоном **извещения об ошибках** как аргументы способа извещения.

$1 Порядковый номер записи в протоколе ошибок

$2 ИД ошибки из записи в протоколе ошибок

$3 Класс из записи в протоколе ошибок

$4 Тип из записи в протоколе ошибок

$5 Значения флагов предупреждений из записи в протоколе ошибок

$6 Имя ресурса из записи в протоколе ошибок

$7 Тип ресурса из записи в протоколе ошибок

$8 Класс ресурса из записи в протоколе ошибок

$9 Метка ошибки из записи в протоколе ошибок

en_name
Однозначно идентифицирует объект. Это уникальное имя используется при удалении объекта.

en_persistenceflg
Указывает, должен ли объект извещения об ошибках автоматически удаляться при повторном запуске системы. Например, чтобы избежать ошибочной генерации сигнала, при повторном запуске системы не должны сохраняться объекты извещения об ошибках, которые содержат действия, отправляющие сигнал другому процессу. При повторном запуске системы процесс, который принимает сигнал, и его идентификатор не сохраняются.

За удаление объекта извещения об ошибках в нужное время отвечает его создатель. В том случае, когда процесс завершается без удаления объекта извещения об ошибках, дескриптор **en_persistenceflg** гарантирует, что при повторном запуске системы устаревшие объекты извещения об ошибках будут удалены.

Допустимы следующие значения дескриптора **en_persistenceflg**:

0 объекты не сохраняются (удаляются во время загрузки)

1 объекты сохраняются (не удаляются во время загрузки)

en_pid
Задает идентификатор процесса (PID), который будет использоваться при идентификации объекта извещения об ошибках. Для объектов с указанным PID дескриптор **en_persistenceflg** должен быть равен 0.

en_rclass
Идентифицирует класс ресурса, в котором происходит сбой. В классе аппаратных ошибок класс ресурса - это класс устройства. В классе программных ошибок это понятие неприменимо.

en_resource
Идентифицирует имя ресурса, в котором происходит сбой. В классе аппаратных ошибок имя ресурса - это имя устройства.

en_rtype
Идентифицирует тип ресурса, в котором происходит сбой. В классе аппаратных ошибок тип ресурса - это тип устройства, заданное в объектном классе "устройства".

en_symptom
Разрешает извещение об ошибке, сопровождаемое строкой признаков, если равен **TRUE**.

en_type
Идентифицирует уровень серьезности записей протокола ошибок, проверяемых на совпадение. Допустимы следующие значения дескриптора **en_type**:

INFO Информационная запись

PEND Угроза потери доступности

PERM Постоянная ошибка

PERF Недопустимое снижение производительности

TEMP Случайная ошибка

UNKN Неизвестная ошибка

Примеры

1. Для создания способа извещения, который будет отправлять пользователю root отформатированную запись об ошибке каждый раз, когда запись об ошибке на диске типа PERM заносится в протокол, создайте файл `/tmp/en_sample.add` со следующим объектом извещения об ошибках:

   ```
   errnotify:
       en_name = "sample"
       en_persistenceflg = 0
       en_class = "H"
       en_type = "PERM"
       en_rclass = "disk"
       en_method = "errpt -a -l $1 | mail -s 'Disk Error' root"
   ```

   Для добавления объекта в класс извещения об ошибках введите:

   ```
   odmadd /tmp/en_sample.add
   ```

   Команда **odmadd** добавляет объект извещения об ошибках, содержащийся в файле **/tmp/en_sample.add**, в файл **errnotify**.

2. Для того чтобы проверить, добавлен ли объект извещения об ошибках в объектный класс, введите:

   ```
   odmget -q"en_name='sample'" errnotify
   ```

   Команда **odmget** находит объект извещения об ошибках с именем **en_name** = `"sample"` в файле **errnotify** и показывает его. Вывод команды выглядит следующим образом:

   ```
   errnotify:
       en_pid = 0
       en_name = "sample"
       en_persistenceflg = 0
   ```

```
    en_label = ""
    en_crcid = 0
    en_class = "H"
    en_type = "PERM"
    en_alertflg = ""
    en_resource = ""
    en_rtype = ""
    en_rclass = "disk"
    en_method = "errpt -a -l $1 | mail -s 'Disk Error' root"
```

3. Для удаления объекта sample из класса объектов Error Notification введите:

   ```
   odmdelete -q"en_name='sample'" -o errnotify
   ```

   Команда **odmdelete** находит объект извещения об ошибках с именем **en_name** = "sample" в файле **errnotify** и удаляет его из объектного класса извещения об ошибках.

4. Для того чтобы отправить пользователю root электронное сообщение в случае повторного возникновения ошибки, создайте файл **/tmp/en_sample.add**, содержащий следующий раздел уведомления об ошибках:

   ```
   errnotify:
           en_name = "errdupxmp"
           en_persistenceflg = 1
           en_dup = "TRUE"
           en_method = "/usr/lib/dupmethod $1"
   ```

   Создайте сценарий **/usr/lib/dupmethod**, как указано ниже:

   ```
   #!/bin/sh
   # Отправка пользователю root электронного сообщения о повторяющейся ошибке.
   # Повторное сообщение из протокола не удаляется.
   #
   # Ввод:
   #   $1 содержится порядковый номер в протоколе.
   #
   # Тело сообщения создается с помощью errpt.
   /usr/bin/errpt -al$1 | /usr/bin/mail -s "Duplicate Error Logged" root >/dev/null

   # Удаление сообщения об ошибке (еще не выполнено)
   #/usr/bin/errclear -l$1 0
   exit $?
   ```

Задачи ведения протокола ошибок

В этом разделе рассмотрены задачи регистрации ошибок.

Следующие разделы посвящены задачам ведения протокола ошибок и содержат информацию по работе со средством ведения протокола:

- Чтение отчета об ошибках
- Примеры подробных отчетов об ошибках
- Пример краткого отчета об ошибках
- Создание отчета об ошибках
- Завершение ведения протокола ошибок
- Очистка протокола ошибок
- Копирование протокола ошибок на дискету или магнитную ленту
- Работа со службами liberrlog

Чтение отчета об ошибках

Для получения отчета о всех ошибках, обнаруженных в течение суток, предшествовавших сбою, введите следующую команду:

```
errpt -a -s ммддччммгг | pg
```

где *ммддччммгг* - месяц, день, час, минута и год суток, предшествовавших сбою.

Отчет об ошибках содержит следующую информацию:

Примечание: Для некоторых ошибок может быть получена не вся информация.

Метка
Предопределенное название события.

идентификатор
Числовой идентификатор события.

Дата/время
Дата и время события.

Порядковый номер
Уникальный номер события.

ИД системы
Идентификатор системного блока.

ИД узла
Мнемоническое имя системы.

Класс
Общий источник ошибки. Существуют следующие классы ошибок:

H Аппаратное обеспечение. (При получении сообщения об ошибке аппаратного обеспечения обратитесь к руководству оператора системы за инструкциями по диагностике отказавшего устройства или другого оборудования. Диагностическая программа определяет состояние устройства, проверяя устройство и анализируя связанные с ним записи протокола ошибок.)

S Программное обеспечение.

O Информационные сообщения.

U Неопределенные (например, сбой сети).

Тип
Серьезность обнаруженной ошибки. Существуют следующие типы ошибок:

PEND Устройство или компонент может стать недоступным.

PERF Производительность устройства или компонента понизилась ниже допустимого уровня.

PERM Неисправимая ошибка. Этот тип относится к самым серьезным ошибкам и свидетельствует о неисправности устройства или модуля программного обеспечения. Ошибки всех типов, кроме PERM, обычно не означают неисправности, но записываются для анализа в диагностических программах.

TEMP Ошибка, которая была исправлена после нескольких неудачных попыток. Этот тип ошибки также применяется для записи информационных сообщений, например, статистики передачи данных устройствами DASD.

UNKN Невозможно определить серьезность ошибки.

INFO Запись протокола ошибок носит информационный характер и не свидетельствует об ошибке.

Имя ресурса
Имя ресурса, обнаружившего ошибку. В случае ошибки программного обеспечения содержит имя компонента программного обеспечения или имя программы. В случае ошибки аппаратного обеспечения - имя устройства или компонента системы. Это не означает, что компонент неисправен и требует замены. Это значение лишь указывает модуль диагностики, применяемый для анализа ошибки.

Класс ресурса
Общий класс ресурса, обнаружившего ошибку (например, класс устройства дисковый накопитель).

Тип ресурса
Тип ресурса, обнаружившего ошибку (например, тип устройства 355mb).

Код расположения
Путь к устройству. Может содержать до четырех полей, соответствующих корпусу, разъему, кабелю и порту.

VPD
Сведения о продукте. В этом поле может быть указана различная информация. Запись об устройстве в протоколе ошибок обычно содержит информацию о производителе устройства, серийном номере, уровнях конструкторских изменений и версиях ПЗУ.

Описание
Краткое описание ошибки.

Возможная причина
Список возможных источников ошибки.

Ошибки пользователя
Список возможных ошибок пользователя, вызвавших сбой. Примером таких ошибок являются неправильно вставленные диски или внешние устройства (такие как модемы и принтеры), питание которых отключено.

Рекомендуемые действия
Инструкции по устранению ошибок, вызванных пользователем.

Ошибка установки
Список возможных ошибок при установке и настройке, вызвавших сбой. Примерами ошибок такого типа являются несовместимость программного и аппаратного обеспечения, неправильное подключение кабелей или их отсоединение, а также неправильно настроенные системы.

Рекомендуемые действия
Инструкции по устранению ошибок, вызванных неправильной установкой.

Причины сбоя
Список возможных неполадок программного и аппаратного обеспечения.

Примечание: Раздел протокола ошибок "возможный сбой" обычно свидетельствует о неполадке программного обеспечения. Если же в протоколе есть записи об ошибке пользователя или установке, но нет записи о возможном сбое, то это обычно означает, что программное обеспечение не является причиной неполадки.

Если вы считаете, что причиной является ошибка программного обеспечения или вам не удается исправить ошибку пользователя или установки, сообщите о неполадке в отдел по обслуживанию программного обеспечения.

Рекомендуемые действия
Инструкции по устранению сбоя. В случае ошибок аппаратного обеспечения список рекомендуемых действий содержит запись ВЫПОЛНИТЕ ПРОЦЕДУРЫ ЛОКАЛИЗАЦИИ НЕПОЛАДКИ. Это значит, что необходимо запустить диагностическую программу.

Подробные сведения

- Уникальные для каждой записи протокола ошибок данные об ошибке, например, код ошибки устройства.
- Информация о текущем рабочем каталоге процесса, например, FILE SYSTEM SERIAL NUMBER (серийный номер файловой системы) и INODE NUMBER (номер узла I) при создании процессом дампа ядра.

Флаг *-A* позволяет просмотреть краткую версию подробного отчета, который выдается, если в команде указан флаг *-a*. Флаг *-A* нельзя применять совместно с флагами *-a*, *-g,* и *-t*. В отчет, создаваемый с помощью флага *-A*, включается следующая информация:

- Метка
- Дата и время
- Тип
- Имя ресурса
- Описание
- Подробные данные

Ниже приведен пример вывода, полученного с помощью этого флага:

```
МЕТКА: STOK_RCVRY_EXIT
Дата/Время:       срд 14 Дек 15.25.33
Тип:              TEMP
Имя ресурса:      tok0
Описание  НЕПОЛАДКА УСТРАНЕНА
Подробные данные FILE NAME строка: 273 файл: stok_wdt.c
SENSE DATA
0000 0000 0000 0000 0000 0000 DEVICE ADDRESS 0004 AC62 25F1
```

Некоторые ошибки можно исключить из отчета. Для просмотра ошибок, исключенных из отчета, введите команду:

```
errpt -t -F report=0 | pg
```

Если такие ошибки есть, включите в отчет все ошибки с помощью команды **errupdate**.

Некоторые ошибки могут не регистрироваться в протоколе. Для просмотра ошибок, исключенных из протокола, введите команду:

```
errpt -t -F log=0 | pg
```

Если такие ошибки есть, включите регистрацию в протоколе для всех ошибок с помощью команды **errupdate**. Регистрация всех ошибок в протоколе необходима для воссоздания ошибки системы.

Примеры подробных отчетов об ошибках

Ниже приведен пример записей отчета об ошибках, созданного с помощью команды **errpt -a**.

Класс ошибки **H** и тип ошибки **PERM** означают, что в системе была обнаружена ошибка устройства (драйвера адаптера SCSI), которую не удалось устранить. С этим типом ошибки могут быть связаны данные диагностики. Они будут показаны в конце сообщения, как показано на следующем примере ошибки драйвера устройства:

```
МЕТКА:      SCSI_ERR1
ИД:         0502F666

Дата/время:       19 июня 22:29:51
Порядковый номер: 95
ИД системы:       123456789012
ИД узла:          host1
Класс:            H
Тип:              PERM
Имя ресурса:      scsi0
Класс ресурса:    adapter
Тип ресурса:      hscsi
Расположение:           00-08
VPD:
     Device Driver Level.........00
     Diagnostic Level............00
     Displayable Message.........SCSI
     EC Level....................C25928
     FRU Number..................30F8834
     Manufacturer................IBM97F
     Part Number.................59F4566
```

```
     Serial Number..............00002849
     ROS Level and ID............24
     Read/Write Register Ptr.....0120

Описание
ADAPTER ERROR

Возможные причины
ADAPTER HARDWARE CABLE
CABLE TERMINATOR DEVICE

Причины сбоя
ADAPTER
CABLE LOOSE OR DEFECTIVE

          Рекомендуемые действия
          PERFORM PROBLEM DETERMINATION PROCEDURES
          CHECK CABLE AND ITS CONNECTIONS

Подробные сведения
SENSE DATA
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Порядковый номер протокола диагностики: 153
Проверенный ресурс:     scsi0
Описание ресурса:       SCSI I/O Controller
Расположение:           00-08
SRN:                    889-191
Описание:               Анализ протокола ошибок указывает на неполадку аппаратного обеспечения.
Возможные FRU:
    SCSI Bus        FRU: n/a            00-08
                    Fan Assembly
    SCSI2           FRU: 30F8834        00-08
                    SCSI I/O Controller
```

Класс ошибки **H** и тип ошибки **PEND** означают, что устройство (Token Ring) может в ближайшее время стать недоступным из-за большого количества ошибок, обнаруженных системой.

```
МЕТКА:    TOK_ESERR
ИД:       AF1621E8

Дата/время:      20 июня 22:28:11
Порядковый номер: 17262
ИД системы:      123456789012
ИД узла:         host1
Класс:           H
Тип:             PEND
Имя ресурса:     TokenRing
Класс ресурса:   tok0
Тип ресурса:     Adapter
Расположение:    TokenRing

Описание
EXCESSIVE TOKEN-RING ERRORS

Возможные причины
TOKEN-RING FAULT DOMAIN

Причины сбоя
TOKEN-RING FAULT DOMAIN

        Рекомендуемые действия
        REVIEW LINK CONFIGURATION DETAIL DATA
        CONTACT TOKEN-RING ADMINISTRATOR RESPONSIBLE FOR THIS LAN

Подробные сведения
SENSE DATA
```

```
0ACA 0032 A440 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 2080 0000 0000 0010 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 78CC 0000 0000 0005 C88F 0304 F4E0 0000 1000 5A4F 5685
1000 5A4F 5685 3030 3030 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000
```

Класс ошибки **S** и тип ошибки **PERM** означают, что в системе была обнаружена ошибка программного обеспечения, которую не удалось устранить.

```
METKA:    DSI_PROC
ИД:       20FAED7F

Дата/время:      Jun 28 23:40:14
Порядковый номер: 20136
ИД системы:      123456789012
ИД узла:         123456789012
Класс:           S
Тип:             PERM
Имя ресурса:     SYSVMM

Описание
Data Storage Interrupt, Processor

Возможные причины
ПРИКЛАДНАЯ ПРОГРАММА

Причины сбоя
ПРИКЛАДНАЯ ПРОГРАММА

        Рекомендуемые действия
        IF PROBLEM PERSISTS THEN DO THE FOLLOWING
        CONTACT APPROPRIATE SERVICE REPRESENTATIVE

Подробные сведения
Data Storage Interrupt Status Register
4000 0000
Data Storage Interrupt Address Register
0000 9112
Segment Register, SEGREG
D000 1018
EXVAL
0000 0005
```

Класс ошибки **S** и тип ошибки **TEMP** означают, что в системе была обнаружена ошибка программного обеспечения. После нескольких попыток системе удалось устранить неполадку.

```
METKA:          SCSI_ERR6
ИД:             52DB7218

Дата/время:      Jun 28 23:21:11
Порядковый номер: 20114
ИД системы:      123456789012
ИД узла:         host1
Класс:           S
Тип:             INFO
Имя ресурса:      scsi0

Описание
SOFTWARE PROGRAM ERROR

Возможные причины
ПРИКЛАДНАЯ ПРОГРАММА

Причины сбоя
ПРИКЛАДНАЯ ПРОГРАММА
```

```
        Рекомендуемые действия
        IF PROBLEM PERSISTS THEN DO THE FOLLOWING
        CONTACT APPROPRIATE SERVICE REPRESENTATIVE

Подробные сведения
SENSE DATA
0000 0000 0000 0000 0000 0011 0000 0008 000E 0900 0000 0000 FFFF
FFFE 4000 1C1F 01A9 09C4 0000 000F 0000 0000 0000 0000 FFFF FFFF
0325 0018 0040 1500 0000 0000 0000 0000 0000 0000 0000 0000 0800
0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000
```

Класс ошибки **O** означает информационное сообщение.

```
METKA:     OPMSG
ИД:        AA8AB241

Дата/время:      Jul 16 03:02:02
Порядковый номер: 26042
ИД системы:      123456789012
ИД узла:         host1
Класс:           O
Тип:             INFO
Имя ресурса:     OPERATOR

Описание
OPERATOR NOTIFICATION

Ошибки пользователя
errlogger COMMAND

        Рекомендуемые действия
        REVIEW DETAILED DATA

Подробные сведения
MESSAGE FROM errlogger COMMAND
hdisk1 : Анализ протокола ошибок указывает на неполадку аппаратного обеспечения.
```

Пример краткого отчета об ошибках

Ниже приведен пример краткого отчета об ошибках, созданного с помощью команды **errpt**. Каждой записи об ошибке соответствует одна строка информации.

```
ERROR_
IDENTIFIER TIMESTAMP  T CL RESOURCE_NAME ERROR_DESCRIPTION
192AC071   0101000070 I 0  errdemon      Error logging turned off
0E017ED1   0405131090 P H  mem2          Memory failure
9DBCFDEE   0101000070 I 0  errdemon      Error logging turned on
038F2580   0405131090 U H  scdisk0       UNDETERMINED ERROR
AA8AB241   0405130990 I O  OPERATOR      OPERATOR NOTIFICATION
```

Создание отчета об ошибках

Выполните следующие действия, чтобы создать отчет об ошибках программного обеспечения или неполадках аппаратного обеспечения:

1. Определите, включено ли ведение протокола ошибок.

   ```
   errpt -a
   ```

   Команда **errpt** создает отчет об ошибках из записей системного протокола ошибок.

   Если протокол ошибок пуст, ведение протокола ошибок было отключено. Активизируйте средство ведения протокола ошибок с помощью следующей команды:

   ```
   /usr/lib/errdemon
   ```

Примечание: Для запуска этой команды необходимы права доступа пользователя root.

Демон **errdemon** запускает ведение протокола ошибок. Если демон не работает, протокол ошибок не ведется.

2. Создайте отчет об ошибках с помощью команды **errpt**. Например, для просмотра всех ошибок дискового накопителя `hdisk1` введите команду:

   ```
   errpt -N hdisk1
   ```

3. Создайте отчет об ошибках с помощью SMIT. Например, с помощью команды **smit errpt**:

   ```
   smit errpt
   ```

   a. Выберите **1**, чтобы направить отчет об ошибках в стандартный вывод, или **2**, чтобы отправить отчет на принтер.

   b. Выберите **да**, чтобы просматривать или распечатывать записи протокола ошибок по мере их добавления. В противном случае выберите **нет**.

   c. Укажите нужное имя устройства в опции **Выбрать имена ресурсов** (например `hdisk1`).

   d. Выберите **Выполнить**.

Завершение ведения протокола ошибок

В данном разделе описано завершение работы средства ведения протокола ошибок.

Для выключения средства ведения протокола ошибок введите команду **errstop**. Для запуска этой команды необходимы права доступа пользователя root.

Как правило, отключать средство ведения протокола ошибок не требуется. Вместо этого следует удалить из протокола ошибок старые и ненужные записи.

Средство ведения протокола ошибок следует отключать при установке или проверке нового программного или аппаратного обеспечения. В этом случае демон ведения протокола ошибок не будет отнимать время центрального процессора на регистрацию известных вам ошибок.

Очистка протокола ошибок

Обычно очистка протокола автоматически выполняется ежедневно с помощью команды **cron**. Если эта процедура не выполняется автоматически, следует время от времени очищать протокол ошибок вручную, предварительно проверив его на наличие записей о серьезных неполадках.

Кроме того, можно удалить записи о конкретных ошибках. Например, после замены дискового накопителя можно удалить из протокола ошибок записи об ошибках старого дискового накопителя.

Для удаления всех записей протокола ошибок выполните одно из следующих действий:

- Вызовите команду **errclear -d**. Например, для удаления всех записей об ошибках программного обеспечения, введите команду:

  ```
  errclear -d S 0
  ```

  Команда **errclear** удаляет из протокола ошибок записи, внесенные раньше определенного числа дней. В предыдущем примере для удаления всех записей указано значение `0`.

- Введите команду **smit errclear**:

  ```
  smit errclear
  ```

Копирование протокола ошибок на дискету или магнитную ленту

Выполните следующие действия, чтобы скопировать протокол ошибок:

- С помощью команд **ls** и **backup** скопируйте протокол ошибок на дискету. Вставьте отформатированную дискету в дисковод и введите команду:

```
ls /var/adm/ras/errlog | backup -ivp
```

- Для копирования протокола ошибок на магнитную ленту вставьте магнитную ленту в накопитель и введите команду:

  ```
  ls /var/adm/ras/errlog | backup -ivpf/dev/rmt0
  ```

- С помощью команды **snap** соберите информацию о конфигурации системы в файл **tar** и скопируйте его на дискету. Вставьте отформатированную дискету в дисковод и введите команду:

  ```
  snap -a -o /dev/rfd0
  ```

  Примечание: Для вызова команды **snap** необходимы права доступа root.

  В этом примере для сбора всей информации о конфигурации системы в команде **snap** указан флаг **-a**. Флаг **-o** позволяет скопировать сжатый файл **tar** на указанное устройство. `/dev/rfd0` указывает дисковод.

  Введите следующую команду, чтобы собрать всю информацию о конфигурации в файле **tar** и скопировать его на магнитную ленту:

  ```
  snap -a -o /dev/rmt0
  ```

  `/dev/rmt0` указывает накопитель на магнитной ленте.

Работа со службами liberrlog

Службы **liberrlog** позволяют читать записи протокола ошибок и обновлять некоторые данные. Они более удобны в применении с языком программирования C, чем со сценариями оболочки. Обращение к протоколу ошибок с помощью функций **liberrlog** намного более эффективно, чем с помощью команды **errpt**.

Информация, связанная с данной:

error_open

errorlog_close

errlog_find, errlog_error_sequence,errlog_find_next

errlog_set_direction

errlog_write

Протокол ошибок и предупреждения

В этом разделе рассмотрен процесс регистрации ошибок и предупреждений.

Если в поле **Предупреждение** шаблона записи об ошибке указано значение `True`, то предупреждения создаются из следующих полей протокола ошибок:

- **Класс**
- **Тип**
- **Описание**
- **Возможная причина**
- **Ошибка пользователя**
- **Ошибка установки**
- **Возможный сбой**
- **Рекомендуемое действие**
- **Подробные данные**

Значения этих полей шаблона должны быть заданы в соответствии с общей архитектурой предупреждения SNA, описание которой находится в книге *Форматы SNA*, номер заказа GA27-3136. С этой книгой можно ознакомиться по адресу: http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/D50A5007. Предупреждения, не соответствующие архитектуре, не могут быть обработаны принимающей программой, например, NetView.

Сообщения, добавляемые в наборы сообщений ведения протокола ошибок, должны соответствовать общей архитектуре предупреждений SNA. При добавлении сообщений с помощью команды **errmsg** выбираются номера сообщений, соответствующих данной архитектуре.

Если в поле **Alert** шаблона записи ошибки указано значение `False`, могут применяться все сообщения из каталога сообщений ведения протокола ошибок.

Управление протоколом ошибок

Для управления средством ведения протокола ошибок применяются команды, процедуры, службы ядра и файлы.

Команды ведения протокола ошибок

`errclear`
Удаляет записи из протокола ошибок. Команда позволяет также стереть весь протокол ошибок. Удаляет записи с указанными номерами ИД ошибки, указанных классов или типов.

`errdead`
Позволяет получить записи об ошибках из буфера **/dev/error** в системном дампе. Системный дамп может содержать записи об ошибках, если перед созданием дампа не работал демон **errdemon**.

`errdemon`
Получает записи об ошибках из файла **/dev/error** и записывает их в протокол ошибок системы. Демон **errdemon** также выполняет функции извещения об ошибках, указанные в объектах извещения об ошибках Администратора объектных данных (ODM). Этот демон автоматически запускается при инициализации системы.

`errinstall`
Позволяет добавлять и заменять сообщения в каталоге сообщений об ошибках. Предназначена для функций установки программного обеспечения. Система создает резервный файл *File*.**undo**. Файл **undo** позволяет отменить изменения, внесенные с помощью команды **errinstall**.

`errlogger`
Заносит в протокол сообщений запись сообщения системного оператора.

`errmsg`
Реализует функции ведения протокола ошибок в пользовательских приложениях. Команда **errmsg** позволяет просмотреть, добавить или удалить сообщения из каталога сообщений об ошибках. Эта команда также позволяет добавить текст в наборы сообщений `Описание ошибки`, `Возможная причина`, `Ошибка пользователя`, `Ошибка установки`, `Возможный сбой`, `Рекомендуемое действие` и `Подробные данные`.

`errpt`
Создает отчет об ошибках из записей системного протокола ошибок. Отчет может быть представлен в виде одной строки данных для каждой записи или полного отчета, включающего все данные, связанные с каждой записью протокола ошибок. Команда позволяет включить в отчет или исключить из него записи различных типов и классов.

`errstop`
Завершает работу демона **errdemon**, который запускается при инициализации системы. Команда **errstop** также отключает некоторые функции диагностики и восстановления системы.

`errupdate`
Позволяет добавить или удалить шаблоны из Хранилища шаблонов записей об ошибках. Позволяет изменить атрибуты шаблона ошибки Предупреждение, Протокол и Отчет. Предназначена для функций установки программного обеспечения.

Функции ведения протокола ошибок и службы ядра

errlog
Записывает сообщение об ошибке в драйвер устройства протокола ошибок.

errsave и errlast
Позволяет ядру и его расширениям заносить записи в протокол ошибок.

errlog_open
Открывает протокол ошибок.

errlog_close
Закрывает протокол ошибок.

errlog_find_first
Ищет первое вхождение записи в протоколе ошибок.

errlog_find_next
Ищет следующее вхождение записи в протоколе ошибок.

errlog_find_sequence
Ищет запись протокола ошибок с указанным порядковым номером

errlog_set_direction
Задает направление поиска в протоколе ошибок

errlog_write
Обновляет запись протокола ошибок.

errresume
Восстанавливает ведение протокола ошибок после команды **errlast**.

Файлы ведения протокола

/dev/error
Обеспечивает интерфейс драйвера стандартного устройства для компонента протокола ошибок.

/dev/errorctl
Обеспечивает интерфейсы драйвера нестандартного устройства для управления системой ведения протокола ошибок.

/usr/include/sys/err_rec.h
Содержит структуры, определенные как аргументы службы ядра **errsave** и функции **errlog**.

/usr/include/sys/errlog.h
Определяет интерфейс процедур **liberrlog**

/var/adm/ras/errlog
Содержит экземпляры сообщений об ошибках и сбоях, обнаруженных в системе.

/var/adm/ras/errtmplt
Содержит Хранилище шаблонов записей об ошибках.

Файловые системы и логические тома

Файл - это одноуровневый массив байтов, представляющих символы ASCII и двоичную информацию.

В AIX файлы могут содержать данные, сценарии оболочки и программы. Кроме того, некоторые файлы представляют собой такие абстрактные объекты, как сокеты и драйверы устройств.

Внутреннее представление файла называется индексным узлом, или *i-узлом*. В журнализированной файловой системе (JFS) i-узел представляет собой структуру, содержащую всю информацию о правах доступа, времени изменения, владельце и расположении данных файла. В JFS размер i-узла составляет 128 байт, а в расширенной журнализированной файловой системе (JFS2) - 512 байт. Кроме того, i-узел содержит адреса

дисковых блоков, в которых хранятся данные файла. I-узел идентифицируется по смещению (*номеру*) и не содержит имени файла. Соответствие между номером i-узла и именем файла называется *связью*.

Имена файлов указываются в каталогах. Каталог - это специальный тип файла, который применяется для поддержки иерархической структуры файловой системы. Каталог состоит из записей. Запись каталога содержит имя файла и номер i-узла.

В данной операционной системе поддерживаются файловые системы JFS и JFS2. Файловая система объединяет данные файлов и каталогов в структуру, которая применяется при чтении и записи данных.

Информация, связанная с данной:

ls

mkfs

pr

fullstat.h

stat

statfs

Типы файлов

Файл - это одномерный массив байтов, имеющий как минимум одну жесткую связь (имя файла). Файлы могут содержать информацию в двоичном или в текстовом (ASCII) виде.

Файлы содержат данные, сценарии оболочки или программы. Кроме того, некоторые файлы представляют такие абстрактные объекты, как сокеты, каналы и драйверы устройств.

В обычных файлах ядро не различает границы записей, поэтому для обозначения границ в программах могут использоваться любые маркеры.

В журнализированных файловых системах (JFS и JFS2) файлы представляются с помощью дисковых индексных узлов (i-узлов). В i-узле хранится информация о файле (принадлежность, режимы доступа, время доступа, адреса данных и время изменения).

Журнализированная файловая система поддерживает следующие типы файлов:

Типы файлов, поддерживаемые журнализированной файловой системой

Тип файла	Имя макрокоманды, используемое в mode.h	Описание
Обычный	**S_ISREG**	Последовательность байтов с одним или несколькими именами. Обычные файлы могут содержать текстовые или двоичные данные. Это файлы прямого доступа; операция чтения или записи может выполняться для любого байта в файле.
Каталог	**S_ISDIR**	Содержит записи каталога (пары "имя файла - i-номер"). Форматы каталогов определяются файловой системой. Процессы могут выполнять чтение каталогов аналогично чтению обычных файлов, однако право записи в каталоги зарезервировано за ядром. Для работы с записями каталогов предназначены специальные функции.
Специальный блоковый файл	**S_ISBLK**	Связывает структурированный драйвер устройства с именем файла.
Специальный символьный файл	**S_ISCHR**	Связывает неструктурированный драйвер устройства с именем файла.

Тип файла	Имя макрокоманды, используемое в mode.h	Описание
Конвейеры	**S_ISFIFO**	Обозначает канал для связи между процессами (IPC). Именованные каналы создаются функцией **mkfifo**. Функция **pipe** создает каналы без имени.
Символьные связи	**S_ISLNK**	Файл, содержащий абсолютный или относительный путь к другому файлу.
Сокеты	**S_ISSOCK**	Механизм IPC для обмена данными между процессами. Для создания сокетов предназначена функция **socket**, а для задания их имен - функция **bind**.

Максимальный размер обычного файла в файловой системе JFS с поддержкой больших файлов составляет около 64 Гб (68 589 453 312). В файловых системах с поддержкой больших файлов и файловых системах JFS других типов максимальный размер файлов, не относящихся в предыдущей таблице к обычным файлам, составляет 2 Гб -1 (2147483647). Максимальный размер файла в JFS2 ограничен размером файловой системы.

Ограничение на размер файловой системы JFS2, установленное в архитектуре, составляет 2^{52} байт (4 петабайта). Максимальный размер файловой системы, поддерживаемый 64-разрядным ядром - 2^{44} - 4096 байт, то есть немного меньше 16 терабайт.

Максимальная длина имени файла - 255 символов, максимальная длина полного имени - 1023 байта.

Работа с файлами

В операционной системе предусмотрено множество функций для работы с файлами. Ниже приведено краткое описание наиболее часто используемых функций:

Функции для создания файлов

Для создания файлов применяются следующие функции:

creat Создает новый пустой обычный файл

link Создает дополнительное имя (запись каталога) для существующего файла

mkdir Создает каталог

mkfifo Создает именованный канал

mknod
Создает файл устройства

open Создает новый пустой файл, если установлен флаг **O_CREAT**

pipe Создает IPC

socket Создает сокет

Функции для управления файлами (программирования)

Для управления файлами предусмотрены следующие функции:

access Определяет, доступен ли файл.

chmod Изменяет режим доступа к файлу.

chown Изменяет принадлежность файла.

close Закрывает дескрипторы открытых файлов (включая сокеты).

fclear Освобождает место в файле.

fcntl, dup или dup2
Управляют дескрипторами открытых файлов.

fsync Записывает внесенные в файл изменения на диск.

ioctl Управляет функциями, связанными с дескрипторами открытых файлов, включая специальные файлы, сокеты и средства поддержки устройств, такие как общий интерфейс терминала termio.

lockf или flock
Управляют дескрипторами открытых файлов.

lseek и llseek
Перемещает указатель ввода/вывода в открытом файле.

open Возвращает дескриптор файла, который используется другими функциями для ссылки на открытый файл. Операция **open** позволяет получить имя файла и значение режима доступа, которое указывает, в каком режиме открывается файл: для чтения, записи или для чтения/записи.

read Считывает данные из открытого файла, если функцией **open** был установлен соответствующий режим доступа (**O_RDONLY** или **O_RDWR**).

rename
Изменяет имя файла.

rmdir Удаляет каталоги из файловой системы.

stat Возвращает информацию о состоянии файла (в том числе имя владельца и режимы доступа).

truncate
Изменяет длину файла.

запись Записывает данные в открытый файл, если функцией **open** был установлен соответствующий режим доступа (**O_WRONLY** или **O_RDWR**).

Дополнительная информация о типах и характеристиках файловых систем приведена в главе Файловые системы в книге *Управление операционной системой и устройствами*.

Работа с каталогами JFS

Каталоги образуют иерархическую структуру, состоящую из файлов, файлов-связей и имен подкаталогов i-узлов. Глубина вложения каталогов не ограничивается. Дисковое пространство выделяется под каталоги блоками по 4096 байт, но операционная система выделяет для каталогов записи по 512 байт.

Процессы могут считывать содержимое каталогов так же, как содержимое обычных файлов. Однако записывать данные в каталоги может только ядро. По этой причине для создания и обслуживания каталогов разработаны специальные функции.

Структура каталогов JFS

Каталог состоит из последовательности записей. Запись каталога содержит три поля фиксированной длины (индекс, связанный с индексным узлом (i-узлом) файла, длину имени файла и количество байт в записи) и одно поле переменной длины для имени файла. Поле имени файла оканчивается символом NULL и дополняется незначащими символами до 4 байт. Максимальная длина имени файла - 255 байт.

Записи каталога имеют переменную длину, что позволяет задавать произвольные имена файлов. Однако объем пространства всех каталогов фиксирован.

Запись каталога не должна занимать более 512 байт. Если каталогу требуется более 512 байт, к первоначальной записи добавляется еще одна запись длиной 512 байт. Если в выделенном блоке данных все 512-байтовые записи заполнены, выделяется дополнительный блок данных (4096 байт).

При удалении файла пространство, занимаемое им в структуре каталога, добавляется к предшествующей записи каталога. Информация об удаленном каталоге хранится до тех пор, пока не будет создана новая запись, которую можно разместить в освободившейся области.

Каждый каталог содержит записи . (точка) и .. (две точки). Знак . (точка) указывает на i-узел самого каталога; запись каталога .. (две точки) - на i-узел родительского каталога. При инициализации файловой системы с помощью программы **mkfs** обе записи . (точка) и .. (две точки) в новом корневом каталоге указывают на корневой i-узел файловой системы.

Для каталогов предусмотрены следующие режимы доступа:

Режим	Описание
read	Процессу разрешено чтение записей каталога
запись	Процессу разрешено создавать новые записи каталога или удалять старые с помощью функций **creat**, **mknod**, **link** и **unlink**
execute	Процессу разрешено использовать данный каталог в качестве текущего рабочего каталога или выполнять поиск в нижестоящем дереве файлов

Работа с каталогами JFS - информация для программистов

Ниже приведен список функций, предназначенных для работы с каталогами:

closedir
Закрывает поток каталога и освобождает структуру, связанную с параметром *указатель-на-каталог*

mkdir Создает каталоги

opendir
Открывает каталог, заданный в параметре *имя-каталога*, и связывает с ним поток каталога

readdir
Возвращает указатель на следующую запись каталога

rewinddir
Устанавливает указанный поток каталога в начало каталога

rmdir Удаляет каталоги

seekdir
Задает позицию для выполнения следующей операции **readdir** в потоке каталога

telldir Возвращает текущее расположение в указанном потоке каталога

Изменение текущего каталога процесса

При загрузке системы текущим каталогом первого процесса становится корневой каталог корневой файловой системы. Новые процессы, создаваемые с помощью функции **fork**, наследуют текущий каталог родительского процесса. Для изменения текущего каталога процесса предназначена функция **chdir**.

Функция **chdir** выполняет синтаксический анализ имени каталога, проверяя, действительно ли целевой объект является каталогом, и есть ли у владельца процесса права доступа к этому каталогу. После выполнения функции **chdir** процесс будет выполнять поиск всех полных имен, не начинающихся с косой черты (/), в новом текущем каталоге.

Изменение корневого каталога процесса

С помощью функции **chroot** можно установить каталог, заданный в параметре процесса *путь*, в качестве действующего корневого каталога. Все дочерние процессы того процесса, который вызвал функцию **chroot**, будут рассматривать указанный каталог как логический корневой каталог файловой системы.

При анализе всех полных имен, начинающихся с / (косой черты), применяется глобальный корневой каталог файловой системы. Это означает, что поиск всех таких имен начинается с этого корневого каталога.

Функции управления каталогами JFS

Файлы каталогов - это уникальные объекты, поэтому для управления каталогами разработаны специальные функции. Они перечислены ниже:

chdir Позволяет перейти в другой каталог

chroot Изменяет текущий корневой каталог

getcwd и getwd
Определяет путь к текущему каталогу

mkdir Создает каталог

opendir,readdir, telldir, seekdir, rewinddir и closedir
Выполняют различные действия над каталогами

rename
Переименовывает каталог

rmdir Удаляет каталог

Работа с каталогами JFS2

Каталоги образуют иерархическую структуру, состоящую из файлов, файлов-связей и имен подкаталогов i-узлов. Глубина вложения каталогов не ограничивается.

Дисковое пространство выделяется под каталоги системными блоками.

Процессы могут считывать содержимое каталогов так же, как содержимое обычных файлов. Однако записывать данные в каталоги может только ядро. По этой причине для создания и обслуживания каталогов разработаны специальные функции.

Структура каталогов JFS2

Каталог состоит из записей, описывающих содержащиеся в нем объекты. Длина записей каталога фиксирована. Запись содержит следующую информацию:

- Номер i-узла
- Имя (длиной не более 22 байт)
- Длину имени
- Дополнительное поле, применяемое в том случае, если имя содержит более 22 байт

Идентификаторы хранятся в каталоге в виде дерева B+, отсортированного по именам. Информация о текущем (.) и родительском (..) каталогах хранится непосредственно в i-узле, а не в записях каталога.

Для каталогов предусмотрены следующие режимы доступа:

Режим	Описание
read	Процессу разрешено чтение записей каталога
запись	Процессу разрешено создавать и удалять записи каталога с помощью функций **creat**, **mknod**, **link** и **unlink**
execute	Процессу разрешено использовать данный каталог в качестве текущего рабочего каталога или выполнять поиск в нижестоящем дереве файлов

Работа с каталогами JFS2 - информация для программистов

Ниже приведен список функций, предназначенных для работы с каталогами:

closedir
Закрывает поток каталога и освобождает структуру, связанную с параметром *указатель-на-каталог*

mkdir Создает каталоги

opendir
Возвращает указатель на структуру, который применяется функцией **readdir** для получения следующей записи каталога; функцией **rewinddir** для перехода к начальной позиции; и функцией **closedir** для закрытия каталога.

readdir
Возвращает указатель на следующую запись каталога

rewinddir
Устанавливает указанный поток каталога в начало каталога

rmdir Удаляет каталоги

seekdir
Устанавливает указатель в позицию, ранее полученную с помощью функции **telldir**

telldir Возвращает текущее расположение в указанном потоке каталога

Функции **open**, **read**, **lseek** и **close** не следует использовать для доступа к каталогам.

Изменение текущего каталога процесса

При загрузке системы текущим каталогом первого процесса становится корневой каталог корневой файловой системы. Новые процессы, создаваемые с помощью функции **fork**, наследуют текущий каталог родительского процесса. Для изменения текущего каталога процесса предназначена функция **chdir**.

Функция **chdir** выполняет синтаксический анализ имени каталога, проверяя, действительно ли целевой объект является каталогом, и есть ли у владельца процесса права доступа к этому каталогу. После выполнения функции **chdir** процесс будет выполнять поиск всех полных имен, не начинающихся с косой черты (/), в новом текущем каталоге.

Изменение корневого каталога процесса

С помощью функции **chroot** процессы могут определять другой корневой каталог. Дочерние процессы вызывающего процесса рассматривают каталог, указанный функцией **chroot**, как логический корневой каталог файловой системы.

При анализе всех полных имен, начинающихся с / (косой черты), применяется глобальный корневой каталог файловой системы. Это означает, что поиск всех таких имен начинается с этого корневого каталога.

Функции управления каталогами JFS2

Файлы каталогов - это уникальные объекты, поэтому для управления каталогами разработаны специальные функции. Они перечислены ниже:

chdir Позволяет перейти в другой каталог

chroot Изменяет текущий корневой каталог

opendir,readdir, telldir, seekdir, rewinddir и closedir
Выполняют различные действия над каталогами

getcwd и getwd
Определяет путь к текущему каталогу

mkdir Создает каталог

rename
Переименовывает каталог

rmdir Удаляет каталог

Работа с i-узлами JFS

Внутри системы файлы журнализированной файловой системы (JFS) представлены как индексные узлы (i-узлы, индексные дескрипторы). I-узлы JFS хранятся в статическом виде на диске и содержат информацию о доступе к файлу, а также указатели на фактические адреса блоков данных файла на диске.

Число i-узлов на диске, доступных файловой системе, зависит от ее размера, от размера группы размещения (по умолчанию 8 Мб) и от количества байтов, отведенных для одного i-узла (по умолчанию 4096). Эти параметры задаются командой **mkfs** при создании файловой системы. Когда число файлов в файловой системе возрастет настолько, что все доступные i-узлы окажутся заняты, новые файлы создаваться не будут, даже если в файловой системе есть свободное место.

Количество доступных i-узлов можно определить с помощью команды **df-v**. Дисковые i-узлы определяются в файле **/usr/include/jfs/ino.h**.

Структура дискового i-узла JFS

Размер дисковых i-узлов в JFS составляет 128 байт. Смещение конкретного i-узла от начала списка i-узлов файловой системы задает уникальный номер (i-номер), по которому операционная система идентифицирует данный i-узел. Для того чтобы в любой момент можно было определить, какие свободные i-узлы доступны файловой системе, создается битовая карта, или *карта i-узлов*.

Дисковые i-узлы содержат следующую информацию:

Поле	Содержимое
i_mode	Тип файла и биты прав доступа
i_size	Размер файла в байтах
i_uid	Права доступа для пользователя с данным ИД
i_gid	Права доступа для группы с данным ИД
i_nblocks	Число блоков, отведенных для размещения файла
i_mtime	Время последнего изменения файла
i_atime	Время последнего обращения к файлу
i_ctime	Время последнего изменения i-узла
i_nlink	Число жестких связей с данным файлом
i_rdaddr[8]	Фактические адреса данных на диске
i_rindirect	Фактический адрес ссылочного блока, если он есть

Содержимое файла нельзя изменить, не изменяя i-узел, однако можно изменить i-узел, не изменяя содержимого файла. Например, при изменении прав доступа будет изменена информация в i-узле (`i_mode`), но данные в файле останутся прежними.

Поле **i_rdaddr** дискового i-узла содержит 8 адресов на диске. Эти адреса указывают на первые 8 блоков данных, связанных с файлом. Адрес в поле **i_rindirect** указывает на ссылочный блок. Ссылочные блоки

могут использовать одноуровневые или двухуровневые ссылки. Таким образом, возможны три варианта выделения блоков для файла: прямое, одноуровневая ссылка и двухуровневая ссылка.

Дисковые i-узлы не содержат ни сокращенных, ни полных имен файлов. Соответствие между именами файлов и i-узлами устанавливается с помощью записей каталогов. Любому i-узлу можно поставить в соответствие несколько имен файлов, создав дополнительные записи каталогов с помощью процедуры **link** или **symlink**. Для того чтобы узнать номер i-узла, соответствующего файлу, нужно ввести команду **ls-i**.

I-узлы, соответствующие файлам устройств, несколько отличаются от i-узлов для обычных файлов. Файлы, связанные с устройствами, называются *особыми файлами*. В особых файлах устройств нет адресов блоков данных, а в поле **i_rdev** указаны основной и дополнительный номера устройства.

Дисковый i-узел освобождается, когда счетчик связей (`i_nlink`) с этим i-узлом становится равен нулю. Связи соответствуют именам файлов, которым назначен данный i-узел. При обнулении счетчика связей на дисковый i-узел все блоки данных, связанные с этим i-узлом, добавляются в битовую карту свободных блоков данных файловой системы. После этого i-узел помещается в схему свободных i-узлов.

Структура базового i-узла JFS

При открытии файла операционная система создает базовый i-узел. *Базовый i-узел* содержит копию всех полей, определенных в дисковом i-узле, а также дополнительные поля для управления доступом к базовому i-узлу. При открытии файла информация из дискового i-узла для упрощения доступа копируется в базовый i-узел. Базовые i-узлы определяются в файле **/usr/include/jfs/inode.h**. Ниже указана некоторая дополнительная информация, хранящаяся в базовом i-узле:

- Состояние базового i-узла, в том числе флаги для индикации:
  - Блокировки i-узла
  - Наличия процесса, ожидающего освобождения i-узла
  - Изменения информации в i-узле данного файла
  - Изменения данных в файле
- Номер логического устройства файловой системы, содержащей файл
- i-номер, служащий для идентификации i-узла
- Счетчик ссылок. Когда счетчик ссылок обнуляется, базовый i-узел освобождается.

При освобождении базового i-узла (например, с помощью функции **close**) счетчик ссылок этого i-узла уменьшается на 1. Если в результате счетчик становится равен нулю, то базовый i-узел освобождается в таблице базовых i-узлов, и его содержимое записывается в копию i-узла, хранящуюся на диске (если две версии i-узла отличаются друг от друга).

Работа с i-узлами JFS2

Внутреннее представление файла в JFS2 называется индексным узлом (или i-узлом).

Дисковые i-узлы JFS2 хранятся в статическом виде на диске и содержат параметры доступа к файлам, а также указатели на фактические адреса блоков данных файлов на диске. I-узлы динамически распределяются файловой системой. Дисковые i-узлы определены в файле **/usr/include/j2/j2_dinode.h**.

При открытии файла операционная система создает базовый i-узел. Этот i-узел содержит копии всех полей, определенных в дисковом i-узле, а также дополнительные контрольные поля. Базовые i-узлы определяются в файле **/usr/include/j2/j2_inode.h**.

Структура дискового i-узла JFS2

Размер дисковых i-узлов в JFS2 составляет 512 байт. Всем i-узлам присвоены уникальные номера, которые хранятся в индексе схемы размещения индексных узлов файловой системы. Эта схема предназначена для контроля за расположением i-узлов на диске и их доступностью.

Дисковые i-узлы содержат следующую информацию:

Поле	Содержимое
di_mode	Тип файла и биты прав доступа
di_size	Размер файла в байтах
di_uid	Права доступа для пользователя с данным ИД
di_gid	Права доступа для группы с данным ИД
di_nblocks	Число блоков, отведенных для размещения файла
di_mtime	Время последнего изменения файла
di_atime	Время последнего обращения к файлу
di_ctime	Время последнего изменения i-узла
di_nlink	Число жестких связей с данным файлом
di_btroot	Корень дерева B+ с фактическими адресами данных на диске

Содержимое файла нельзя изменить, не изменяя i-узел, однако можно изменить i-узел, не изменяя содержимого файла. Например, при изменении прав доступа будет изменена информация в i-узле (**di_mode**), но данные в файле останутся прежними.

Узел **di_btroot** соответствует корню дерева B+. Он содержит описание данных, хранящихся в i-узлах. Одно из полей дерева di_btroot указывает, как много записей в его i-узле уже используется, а другое поле указывает тип этих записей - листья или внутренние узлы дерева B+.

Дисковые i-узлы не содержат ни сокращенных, ни полных имен файлов. Соответствие между именами файлов и i-узлами устанавливается с помощью записей каталогов. Любому i-узлу можно поставить в соответствие несколько имен файлов, создав дополнительные записи каталогов с помощью процедуры **link** или **symlink**. Для того чтобы узнать номер i-узла, соответствующего файлу, нужно ввести команду **ls-i**.

I-узлы, соответствующие файлам устройств, несколько отличаются от i-узлов для обычных файлов. Файлы, связанные с устройствами, называются *особыми файлами*. В особых файлах устройств нет адресов блоков данных, а в поле **di_rdev** указаны основной и дополнительный номера устройства.

Дисковый i-узел освобождается, когда счетчик связей (**di_nlink**) с этим i-узлом становится равен 0. Связи представляют имена файлов, которым назначен данный i-узел. При обнулении счетчика связей на дисковый i-узел все блоки данных, связанные с этим i-узлом, добавляются в битовую карту свободных блоков данных файловой системы. После этого i-узел помещается в схему свободных i-узлов.

Структура базового i-узла JFS2

При открытии файла информация из дискового i-узла для упрощения доступа копируется в базовый i-узел. Структура базового i-узла содержит дополнительные поля, предназначенные для управления доступом к важным данным i-узла. Поля базового i-узла определяются в файле **j2_inode.h**. Ниже указана некоторая дополнительная информация, хранящаяся в базовом i-узле:

- Состояние базового i-узла, в том числе флаги для индикации:
  - Блокировки i-узла
  - Наличия процесса, ожидающего освобождения i-узла
  - Изменения информации в i-узле данного файла
  - Изменения данных в файле
- Номер логического устройства файловой системы, содержащей файл
- i-номер, служащий для идентификации i-узла
- Счетчик ссылок. Когда счетчик ссылок обнуляется, базовый i-узел освобождается.

При освобождении базового i-узла (например, с помощью функции **close**) счетчик ссылок этого i-узла уменьшается на 1. Если в результате счетчик становится равен нулю, то базовый i-узел освобождается в таблице базовых i-узлов, и его содержимое записывается в копию i-узла, хранящуюся на диске (если две версии i-узла отличаются друг от друга).

Выделение памяти для файлов в JFS

Способ выделения памяти для файлов - это способ, применяемый операционной системой для размещения данных в физической памяти.

Ядро выделяет дисковую память для файлов и каталогов в виде логических блоков. В JFS *логическим блоком* называется блок данных файла или каталога размером 4096 байт. В действительности логические блоки не создаются на диске, однако для относящихся к ним данных выделяется дисковая память. Любой файл или каталог состоит из логических блоков. В JFS память выделяется не логическими блоками, а фрагментами.

Полные и неполные логические блоки

Файл или каталог может занимать весь логический блок или только его часть. Полный логический блок содержит 4096 байт данных. Неполный логический блок образуется тогда, когда размер файла или каталога не кратен 4096 байтам.

Например, файл размером 8192 байт займет два логических блока. В каждом из них будет содержаться 4096 байт данных. В то же время файл размером 4608 тоже будет состоять из двух логических блоков. При этом последний логический блок будет неполным, так как он будет содержать лишь 512 байт данных файла. Неполным может быть только последний логический блок файла.

Выделение памяти во фрагментированной файловой системе

По умолчанию размер фрагмента равен 4096 байт. При создании файловой системы можно указать меньший размер фрагмента в команде **mkfs**. Допустимы фрагменты размером 512, 1024, 2048 и 4096 байт. Во всей файловой системе применяется единый размер фрагмента.

Для более эффективной работы с файлами и каталогами размером более 32 Кб для них выделяются фрагменты размером по 4096 байт. Такой фрагмент содержит полный логический блок. При добавлении данных в файл или каталог ядро выделяет для размещения логических блоков дополнительные фрагменты дискового пространства. Таким образом, если в файловой системе размер фрагмента равен 512 байтам, то для размещения полного логического блока потребуется восемь фрагментов.

Дисковое пространство выделяется ядром таким образом, что неполный блок всегда является последним. Если размер неполного блока превысит размер выделенного дискового пространства, то для этого блока будут выделены дополнительные фрагменты. Когда размер неполного блока достигает 4096 байт, его данные перемещаются в один фрагмент размером 4096 байт. Если размер неполного блока меньше 4096 байт, то его данные размещаются в нескольких фрагментах, суммарный размер которых наиболее точно соответствует размеру блока.

Перемещение фрагментов осуществляется также при добавлении данных в логический блок, соответствующий пустому пространству в файле. *Пустое пространство в файле* - это пустой логический блок, расположенный перед последним логическим блоком с данными. (В каталогах пустых пространств не бывает.) Пустые блоки не записываются на диск. Фрагменты для такого блока выделяются только при заполнении пустого пространства данными. Для логического блока, который ранее не был записан на диск, выделяется фрагмент размером 4096 байт.

При записи данных поверх старых данных файла или каталога дополнительные блоки не выделяются. Для логического блока, содержавшего эти данные, фрагменты уже были выделены ранее.

Для размещения логических блоков файла или каталога JFS всегда пытается выделить непрерывную область дисковой памяти. Это позволяет сократить время поиска, так как данные файла или каталога размещаются в одной области памяти и их можно считывать последовательно. Однако для разных логических блоков не всегда выделяются смежные фрагменты диска. Это связано с тем, что область диска, смежная с одним из фрагментов, может быть уже занята другим файлом или каталогом. Однако фрагменты одного логического блока всегда образуют непрерывную область памяти.

Для хранения информации о состоянии фрагментов в файловой системе применяется битовая таблица, которая называется *картой размещения блоков*. С помощью этой карты файловая система ищет свободные фрагменты, которые могут быть выделены для размещения данных файла. В каждый момент времени фрагмент может принадлежать только одному файлу или каталогу.

Выделение памяти в файловой системе JFS со сжатием данных

В файловой системе со сжатием данных дисковое пространство выделяется для каталогов. Сжатие данных выполняется только для обычных файлов и символьных связей, размер которых превосходит значение, указанное в i-узле.

В файловых системах со сжатием данных дисковое пространство выделяется так же, как и во фрагментированных файловых системах. При изменении логического блока ему выделяется 4096 байт дискового пространства. Выделение такого объема памяти гарантирует, что данные логического блока будут размещены на диске даже без сжатия. В этой файловой системе первая операции записи в логический блок файла должна возвращать код ошибки "Недостаточно дисковой памяти". После выполнения операции данные логического блока сжимаются, и лишь затем записываются на диск. При этом для хранения логического блока будет выделено ровно столько фрагментов, сколько необходимо.

Во фрагментированной файловой системе только последний логический блок файла, размер которого меньше 32 Кб, размещается во фрагменте размером меньше 4096 байт. Такой логический блок называется неполным. В файловой системе со сжатием данных неполными могут быть все логические блоки.

После записи логического блока на диск он больше не считается измененным. При каждом последующем изменении логического блока ему снова будет выделяться полный блок диска в соответствии с правилами файловой системы. После записи логического блока со сжатыми данными на диск исходный полный фрагмент будет освобождаться.

Выделение памяти в файловой системе JFS с поддержкой больших файлов

В файловой системе JFS с поддержкой больших файлов для обычных файлов выделяются фрагменты двух типов. "*Большие*" фрагменты (32 x 4096) выделяются для логических блоков, размер которых превышает 4 Мб. Для всех остальных логических блоков выделяются фрагменты размером 4096 байт. Особые файлы размещаются во фрагментах размером 4096 байт. В такой файловой системе максимальный размер файла составляет около 64 Гб (68 589 453 312).

Большой фрагмент представляет собой 32 смежных фрагмента по 4096 байт. В связи с этим рекомендуется, чтобы в файловых системах с поддержкой больших файлов в основном хранились именно большие файлы. Хранение большого числа маленьких файлов (размер которых меньше 4 Мб) может привести к высокому уровню фрагментации дискового пространства. Это может привести к тому, что запросы на выделение большой области памяти будут отклоняться с кодом ошибки ENOSPC, так как в файловой системе не будет 32 свободных смежных фрагментов.

Формат адреса дисковой памяти

Для работы с фрагментами JFS необходима адресация на уровне фрагментов. Адреса дисковой памяти задаются в специальном формате, позволяющем определять расположение фрагментов логического блока на диске. Во фрагментированных файловых системах и в файловых системах со сжатием данных

применяется одинаковый формат адресов дисковой памяти. Эти адреса хранятся в поле **i_rdaddr** i-узлов и в косвенных блоках. Каждый адрес указывает на непрерывную последовательность фрагментов на диске.

Адрес дисковой памяти содержит два поля: **nfrags** и **addr**. Эти поля описывают область диска, с которой связан адрес:

addr
Задает начальный фрагмент последовательности.

nfrags
Задает общее число смежных фрагментов, не используемых данным адресом.

Например, если в файловой системе размер фрагмента равен 512 байт, а логический блок занимает 8 фрагментов, то значение **nfrags**, равное 3, указывает, что адрес идентифицирует 5 фрагментов.

Ниже приведены примеры значений **addr** и **nfrags** для различных адресов дисковой памяти. В этих примерах предполагается, что размер фрагмента равен 512 байт, и логический блок занимает 8 фрагментов.

Адрес одного фрагмента:

```
addr:   143
nfrags: 7
```

Этот адрес указывает, что данные располагаются на диске, начиная со 143 фрагмента. Значение **nfrags** указывает, что адрес ссылается только на один фрагмент. В файловой системе с другим размером фрагментов значение **nfrags** было бы другим. Для правильной интерпретации значения **nfrags** система или пользователь, вычисляющий адрес, должны знать размер фрагмента в файловой системе.

Адрес пяти фрагментов:

```
addr:   1117
nfrags: 3
```

В данном случае адрес задает последовательность фрагментов, которая начинается со 1117 фрагмента на диске и содержит 5 фрагментов (с учетом начального). Значение **nfrags** указывает, что адрес не ссылается на три фрагмента логического блока.

Размер адреса дисковой памяти представляет собой 32-разрядное число. Биты нумеруются от 0 до 31. Нулевой бит всегда зарезервирован. Биты 1 - 3 содержат значение **nfrags**. Биты 4 - 31 содержат значение **addr**.

Косвенные блоки JFS

Косвенные блоки JFS позволяют идентифицировать участки дискового пространства, выделенные для размещения больших файлов. Применение косвенной адресации обеспечивает поддержку файлов различных размеров и повышает эффективность работы с файлами. Косвенный блок указывается в поле **i_rindirect** i-узла. Существуют следующие способы адресации фрагментов диска, в которых значение этого поля интерпретируется по-разному:

- Прямая
- Одноуровневая ссылка
- Двухуровневая ссылка

Во всех указанных способах используется тот же формат адреса дисковой памяти, что и во фрагментированной файловой системе. Поскольку для файлов размером больше 32 Кб выделяются фрагменты размером 4096 байт, то поле **nfrags** в адресах, задаваемых в формате одноуровневой или двухуровневой ссылки, равно 0.

Метод прямой адресации

В этом методе адресации поле **i_rdaddr** i-узла содержит до 8 адресов фрагментов файла. Максимальный размер файла, размещение которого может быть описано способом прямой адресации, составляет 32 768 байт (32 Кб), т.е. 8 x 4096 байт. Если размер файла превышает 32 Кб, то для описания его размещения на диске применяется косвенная адресация.

Метод одноуровневых ссылок

В этом методе поле **i_rindirect** содержит адрес косвенного блока первого уровня. Косвенный блок, адрес которого указан в поле **i_rindirect**, содержит 1024 адреса. Каждый из этих адресов идентифицирует фрагмент диска. Максимальный размер файла, размещение которого может быть описано таким способом, составляет 4 194 304 байт (4 Мб), или 1024 x 4096 байт.

Метод двухуровневых ссылок

В этом методе в поле **i_rindirect** указывается адрес косвенного блока второго уровня. Этот блок содержит 512 адресов, указывающих на косвенные блоки первого уровня, в которых хранятся адреса фрагментов файла. Максимальный размер файла, размещение которого может быть описано с помощью такого метода в файловой системе без поддержки больших файлов, составляет 2 147 483 648 байт (2 Гб), или 512 x (1024 x 4096) байт.

Примечание: Максимальный размер файла, который поддерживается системными вызовами **read** и **write**, составляет 2 Гб - 1 (2^{31}–1). Если используется интерфейс прямого доступа, можно обращаться к файлам размером 2 Гб.

В файловых системах с поддержкой больших файлов допускается хранение файлов, размер которых почти достигает 64 Гб (68 589 453 312). В этом случае первый косвенный блок первого уровня содержит адреса фрагментов размером 4096 байт, а все остальные косвенные блоки первого уровня - адреса фрагментов размером 32 X 4096 байт. Максимальный размер файла в файловой системе с поддержкой больших файлов вычисляется по следующей формуле:

```
(1 * (1024 * 4096)) + (511 * (1024 * 131072))
```

Для каталогов выделяются фрагменты размером 512 байт. При добавлении данных в каталог для него выделяются дополнительные фрагменты того же размера.

Квоты

Квоты на дисковую память задают объем пространства файловой системы, который может быть предоставлен конкретному пользователю или группе.

Функция **quotactl** устанавливает ограничение на число файлов и число блоков диска, которые могут быть выделены в файловой системе одному пользователи или группе. Существуют ограничения двух типов:

hard Задает максимальное значение. Если объем дисковой памяти, выделенной процессу, достиг жесткого ограничения, то запрос на выделение дополнительной памяти будет отклонен.

soft Фактическое ограничение. Если объем памяти, выделенной процессу, достигает гибкого ограничения, то на терминал пользователя выводится предупреждение. Часто оно появляется при входе пользователя в систему. Если пользователь не устранил ошибку в течение нескольких сеансов работы, гибкое ограничение может стать жестким.

Предупреждения системы уведомляют пользователей о превышении гибких квот. Однако фактически процесс может превысить гибкое ограничение, если общий объем захваченных им ресурсов не превышает жесткого ограничения.

Выделение памяти для файлов в JFS2

Способ выделения памяти для файлов - это способ, применяемый операционной системой для размещения данных в физической памяти.

Ядро выделяет дисковую память для файлов и каталогов в виде *логических блоков*. Логическим блоком называется блок данных файла или каталога размером 512, 1024, 2048 или 4096 байт. Размер логического блока указывается при создании файловой системы JFS2. В действительности логические блоки не создаются на диске, однако для относящихся к ним данных выделяется дисковая память. Любой файл или каталог состоит из логических блоков.

Полные и неполные логические блоки

Файл или каталог может занимать весь логический блок или только его часть. Логический блок может содержать 512, 1024, 2048 или 4096 байт данных, в зависимости от размера блока, указанного при создании файловой системы JFS2. Неполный логический блок образуется тогда, когда размер файла или каталога не кратен размеру блока файловой системы.

Например, в файловой системе JFS2 с размером блока 4096 байт файл размером 8192 байта будет занимать два логических блока. В каждом из них будет содержаться 4096 байт данных. В то же время файл размером 4608 тоже будет состоять из двух логических блоков. При этом второй логический блок будет содержать лишь 512 байт данных файла.

Распределение памяти в JFS2

По умолчанию размер блока равен 4096 байтам. При создании файловой системы можно указать другой размер блока в команде **mkfs**. Допустимы блоки размером 512, 1024, 2048 и 4096 байт. Во всей файловой системе применяется единый размер блока.

Ядро выделяет дисковую память таким образом, что только последний блок данных файла будет заполнен не полностью. После заполнения этого блока будет выделен новый.

Перемещение фрагментов осуществляется также при добавлении данных в логический блок, соответствующий пустому пространству в файле. *Пустое пространство в файле* - это пустой логический блок, расположенный перед последним логическим блоком с данными. (В каталогах пустых пространств не бывает.) Пустые блоки не записываются на диск. Фрагменты для такого блока выделяются только при заполнении пустого пространства данными. Для логического блока, который ранее не был записан на диск, выделяется полный блок файловой системы.

При записи данных поверх старых данных файла или каталога дополнительные блоки не выделяются. Для логического блока, содержавшего эти данные, блоки файловой системы уже были выделены ранее.

Для размещения логических блоков файла или каталога JFS2 всегда пытается выделить непрерывную область дисковой памяти. Это позволяет сократить время поиска, так как данные файла или каталога размещаются в одной области памяти и их можно считывать последовательно. Это связано с тем, что область диска, смежная с одним из фрагментов, может быть уже занята другим файлом или каталогом.

Для хранения информации о состоянии фрагментов в файловой системе применяется битовая таблица, которая называется *картой размещения блоков*. С помощью этой карты файловая система ищет свободные блоки, которые могут быть выделены для размещения данных файла. В каждый момент времени блок может принадлежать только одному файлу или каталогу.

Области

Область - это непрерывная последовательность блоков файловой системы, целиком выделенная объекту JFS2. Большие области могут находиться в нескольких группах размещения.

I-узел представляет любой объект JFS2. I-узлы содержат такую информацию, как даты, связанные с файлом, тип файла (обычный или каталог и пр.) Кроме того, они содержат двоичное дерево, описывающее расположение областей.

Для определения области нужно указать ее адрес и длину. Длина измеряется в блоках файловой системы. Для хранения длины области отведено 24 разряда, поэтому область может содержать от 1 до 2^{24} -1 блоков файловой системы. Таким образом, максимальный размер области зависит от размера блока файловой системы. Адрес области - это адрес первого блока области. Адрес также измеряется в блоках файловой системы от ее начала.

Расширенная файловая система в сочетании с пользовательским размером блока позволяет системе JFS2 не обрабатывать отдельно внутреннюю фрагментацию. Вы можете указать небольшой размер блока (например, 512 байт) для минимизации внутренней фрагментации при большом количестве небольших файлов.

Стратегия выделения памяти JFS2 в целом направлена на наиболее последовательное выделение памяти путем минимизации числа областей. Это увеличивает производительность за счет подсистемы ввода-вывода. Однако размещение всех файлов в одном блоке не всегда возможно.

Двоичные (b+) деревья

Структура файлов представляет собой двоичное дерево. Чаще всего в JFS2 выполняются операции чтения и записи областей. Для повышения производительности этих операций применяются деревья B+.

Описатель размещения областей (структура **xad_t**) описывает область и содержит два дополнительных поля: логическое смещение области в файле и поле флагов. Структура **xad_t** определена в файле **/usr/include/j2/j2_xtree.h**.

Структура **xad** описывает два абстрактных диапазона:

- Физический диапазон блоков диска. Этот диапазон начинается с блока с номером addressXAD(xadp) и имеет длину lengthXAD(xadp).
- Логический диапазон байт в файле. Этот диапазон начинается с байта с номером offsetXAD(xadp)*(размер блока файловой системы) и имеет длину lengthXAD(xadp)*(размер блока файловой системы).

Физический диапазон и логический диапазон имеют одинаковую длину в байтах. Обратите внимание, что смещение измеряется в блоках файловой системы; например, 3 означает 3 блока файловой системы, а на 3 байта. Области файла всегда выровнены по размеру блока файловой системы.

Ограничение JFS2

Для расширения файлов JFS2 требуется смежная область памяти размером не менее одной страницы (4 Кб). Если смежная область памяти размером не менее 4 Кб отсутствует, расширение файлов в файловой системе будет запрещено даже в том случае, если в меньших блоках доступен достаточный объем памяти.

Структура файловой системы JFS

Файловая система - это набор файлов, каталогов и других структур.

Файловые системы предназначены для хранения информации и распределения по дискам данных файлов и каталогов. Помимо файлов и каталогов, файловые системы JFS включают загрузочный блок, главный блок, битовые образы и одну или несколько групп размещения. Каждая файловая система занимает один логический том.

Загрузочный блок JFS

Загрузочный блок занимает на диске первые 4096 байт файловой системы, начиная с байта со смещением 0. Этот блок может применяться для запуска операционной системы.

Главный блок JFS

Главный блок занимает 4096 байт на диске и располагается, начиная с байта со смещением 4096. В следующих полях главного блока хранится информация обо всей файловой системе:

- Размер файловой системы
- Число блоков данных в файловой системе
- Флаг состояния файловой системы
- Размеры групп размещения

Битовые карты размещения JFS

В файловой системе предусмотрены следующие битовые карты размещения:

- Карта размещения фрагментов, которая содержит записи о состоянии размещения всех фрагментов.
- Карта размещения дисковых i-узлов, которая содержит информацию о состоянии всех i-узлов.

Фрагменты JFS

Многие файловые системы организованы в виде блоков дисковой памяти или блоков данных. В этом случае диск разделяется на блоки равного размера; данные хранятся в файлах или логических блоках каталогов. В свою очередь, блок на диске можно разделить на более мелкие единицы фиксированного размера, называемые *фрагментами*. В некоторых системах фрагменты не могут выходить за границы дискового блока. Другими словами, логический блок не может содержать фрагменты из других дисковых блоков.

Однако, в журнализированной файловой системе (JFS) файловая система рассматривается как непрерывная последовательность фрагментов. Фрагмент JFS представляет собой основную единицу выделения памяти, а адресация на диске реализована на уровне фрагментов. Следовательно, фрагменты могут выходить за границы возможных дисковых блоков. По умолчанию размер фрагмента JFS равен 4096 байт, хотя можно задать и меньший размер. Помимо данных для файлов и каталогов, фрагменты содержат также дисковые адреса и данные для косвенных блоков.

Группы размещения JFS

Множество фрагментов, образующих файловую систему, разбивается на один или несколько блоков фиксированного размера, объединяющих смежные фрагменты. Эти блоки называются *группой размещения*. Первая группа начинает файловую систему и включает резервную область, которая занимает первые 32 x 4096 байт в группе. Первые 4096 байт этой области выделяются под загрузочный блок, следующие 4096 байт - под главный блок файловой системы.

Каждая группа размещения содержит постоянное число смежных дисковых i-узлов, занимающих несколько фрагментов группы. Эти фрагменты резервируются под i-узлы при создании и расширении файловой системы. В первой группе размещения дисковые i-узлы занимают фрагменты, следующие непосредственно за резервной областью. В следующих областях дисковые i-узлы располагаются в начале каждой группы. Дисковые i-узлы имеют размер 128 байт и идентифицируются уникальным номером дискового i-узла или просто i-номером. i-номер задает расположение дискового i-узла на диске или в группе размещения.

Размер группы размещения фрагментов и размер группы размещения дисковых i-узлов определяется как число фрагментов и дисковых i-узлов, существующих в каждой группе размещения. По умолчанию размер группы размещения составляет 8 Мб. Максимальный размер группы размещения равен 64 Мб. Эти значения хранятся в главном блоке файловой системы и задаются при ее создании.

Применение групп размещения позволяет использовать в стратегиях выделения ресурсов JFS эффективные методы повышения производительности при выполнении операций ввода/вывода в файловой системе. Эти стратегии основаны на объединении блоков и дисковых i-узлов, содержащих связанные данные, в кластеры, что позволило бы достичь оптимального размещения данных на диске. Чтение и запись файлов часто происходит последовательно, и файлы в одном каталоге часто используются один за другим. Кроме того, стратегии размещения распределяют несвязанные данные в файловой системе таким образом, чтобы минимизировать фрагментацию свободного дискового пространства.

Дисковые i-узлы JFS

Данные файла или каталога хранятся в логическом блоке пакетами по 4096 байт. Для хранения этих данных в логическом блоке выделяются фрагменты. Каждому файлу и каталогу соответствует i-узел, содержащий информацию о доступе, такую как тип файла, права доступа, ИД владельца и число связей с данным файлом. Кроме того, эти i-узлы содержат "адреса", позволяющие находить на диске данные логических блоков.

i-узел состоит из последовательности пронумерованных разделов. Каждый раздел содержит адрес одного из логических блоков файла или каталога. Эти адреса указывают начальный фрагмент и полное число фрагментов в блоке. Например, файл с размером 4096 байт имеет в массиве i-узла один адрес. Эти 4096 байт данных хранятся в одном логическом блоке. Файл с размером 6144 байта имеет два адреса. Один адрес указывает на первые 4096 байт, а второй - на оставшиеся 2048 байт (частичный логический блок). Если файл занимает большое число логических блоков, то в i-узел не записываются адреса дисковой памяти. Вместо этого в i-узел помещается ссылка на косвенный блок, содержащий дополнительные адреса.

Структура файловой системы JFS2

Файловая система - это набор файлов, каталогов и других структур.

Файловые системы предназначены для хранения информации и распределения по дискам данных файлов и каталогов. Помимо файлов и каталогов, файловые системы содержат главный блок, карты размещения, а также одну или несколько групп размещения. Каждая файловая система занимает один логический том.

Главный блок JFS2

Главный блок занимает 4096 байт на диске и располагается начиная с байта со смещением 32768. В следующих полях главного блока хранится информация обо всей файловой системе:

- Размер файловой системы
- Число блоков данных в файловой системе
- Флаг состояния файловой системы
- Размеры групп размещения
- Размер блока файловой системы

Карты размещения JFS2

В файловой системе предусмотрены следующие карты размещения:

- Карта размещения i-узлов содержит информацию о расположении и состоянии всех i-узлов файловой системы.
- Карта размещения блоков содержит расположение и состояние всех блоков файловой системы.

Дисковые i-узлы JFS2

Логический блок содержит данные одного файла или каталога. Дисковая память логического блока выделяется блоками файловой системы. Каждому файлу и каталогу соответствует i-узел, содержащий

информацию о доступе, такую как тип файла, права доступа, ИД владельца и число связей с данным файлом. Кроме того, i-узлы содержат двоичное (B+) дерево, позволяющее определить расположение данных логического блока на диске.

Группы размещения JFS2

Группы размещения делят пространство файловой системы на участки. Они относятся к способу решения задач, при котором для следующего шага выбирается наиболее подходящее решение из нескольких опробованных решений. Группы размещения позволяют использовать стандартные способы оптимизации производительности ввода-вывода в стратегиях размещения ресурсов JFS2. Вначале стратегия размещения пытается объединить блоки и дисковые i-узлы, содержащие связанные данные, в кластеры, что позволило бы достичь оптимального размещения данных на диске. Чтение и запись файлов часто происходит последовательно, и файлы в одном каталоге часто используются один за другим. Кроме того, стратегии размещения распределяют несвязанные данные в файловой системе таким образом, чтобы минимизировать фрагментацию свободного дискового пространства.

Группы размещения файловой системы идентифицируются по индексу, или номеру группы (начинается с нуля).

Размеры групп размещения должны выбираться таким образом, чтобы обеспечить последовательный доступ к большому объему данных. Всего может существовать не более 128 групп размещения. Минимальный размер группы размещения - 8192 блока файловой системы.

Неполные группы размещения

Файловая система, размер которой не кратен размеру группы размещения, содержит одну неполную группу размещения; последняя группа размещения файловой системы содержит меньшее число блоков, чем остальные. Неполная группа размещения ничем не отличается от обычной, кроме того, что несуществующие блоки в ее карте размещения помечены как занятые.

Работа с большими файлами

В AIX поддерживаются файлы размером более 2 Гб. В этом разделе описаны особенности больших файлов, которые программист должен учитывать при разработке приложений. С помощью ряда программных интерфейсов существующие приложения можно изменить таким образом, чтобы они поддерживали большие файлы. Интерфейсы файловой системы обычно основаны на типе данных **off_t**.

Особенности работы старых программ

32-разрядная среда, применявшаяся всеми приложениями в версиях младше AIX 4.2, осталась без изменений. Однако старые приложения не поддерживают обработку больших файлов.

Например, поле **st_size** структуры **stat**, в котором программе возвращается размер файла, представляет собой 32-разрядное целое число со знаком. В связи с этим, структура **stat** непригодна для работы с файлами, размер которых превышает LONG_MAX. Если приложение попытается вызвать функцию **stat** для файла, размер которого больше LONG_MAX, то в функции **stat** произойдет ошибка EOVERFLOW, означающая, что размер файла не умещается в соответствующем поле структуры.

Это может приводит к ошибкам в программах, не рассчитанных на работу с большими файлами, даже если эти программы не будут пытаться выполнять какие-либо операции с такими файлами.

Ошибка EOVERFLOW может возникать и при выполнении функций **lseek** и **fcntl**, если возвращаемые ими значения превосходят размер типа данных или структуры, применяемой программой. В функции **lseek** ошибка EOVERFLOW будет возникать в тех случаях, когда смещение превышает LONG_MAX. В функции **fcntl** ошибка EOVERFLOW может возникать при попытке выполнения операции F_GETLK, если смещение или длина блокируемой области превышает значение LONG_MAX.

Защита открытых файлов

Использование существующих программ для работы с большими файлами может повлечь за собой непредсказуемые результаты, включая потерю данных. Для защиты приложений от таких сбоев в AIX предусмотрены средства защиты, запрещающие старым программам выполнять операции, которые могут повредить большие файлы.

В операционной системе предусмотрены и другие механизмы защиты больших файлов. Суть этих механизмов сводится к тому, что старые программы работают в той же среде, что и ранее, не имея возможности повредить большие файлы. Например, если приложение попытается записать в файл более 2 ГБ данных с помощью функций семейства **write**, то будет записано только 2 ГБ - 1 байт. Если приложение попытается превысить это ограничение, то функция **write** выдаст ошибку EFBIG. Функции **mmap**, **ftruncate** и **fclear** работают аналогично.

Функции семейства **read** также снабжены механизмами защиты. Если приложение попытается считать более 2 Гб данных, то будет прочитано только 2 Гб минус 1 байт. Если приложение явно запросит данные, расположенные за этой границей, то будет выдана ошибка EOVERFLOW.

Для защиты открытых файлов в описание открытого файла теперь добавлен специальный флаг. Текущее состояние этого флага можно определить с помощью команды F_GETFL процедуры **fcntl**. Изменить его состояние можно с помощью команды F_SETFL процедуры **fcntl**.

Поскольку описания открытых файлов наследуются в семействе функций **exec**, то программы, передающие описания больших файлов другим программам, должны проверять, могут ли принимающие программы правильно работать с такими файлами.

Модификация программ для работы с большими файлами

В AIX предусмотрено два способа добавления поддержки больших файлов в существующие приложения. Программист может выбрать любой из них, по своему усмотрению:

- Первый из них заключается в добавлении определения _LARGE_FILES, которое правильно переопределяет все зависимые типы данных и структуры, а вместо старых функций подставляет новые функции, поддерживающие большие файлы. Этот способ сохраняет переносимость программы, так как она по-прежнему соответствует стандартам POSIX и XPG. Недостаток этого способа заключается в том, что за счет автоматического переопределения типов данных и подстановки новых функций текст программы перестает отражать фактические типы данных.
- Второй способ заключается в явной замене старых функций на функции, поддерживающие большие файлы. Изменение кода программы требует больше усилий и снижает возможность переноса программы на другие платформы. Этот подход целесообразен только в тех случаях, когда определение символа _LARGE_FILES нежелательно по каким-либо причинам, и изменения затрагивают только малую часть текста программы.

В любом случае приложение *необходимо* тщательно проверить на его совместимость с большими файлами.

Определение _LARGE_FILES

В стандартной среде компиляции тип данных **off_t** определен как 32-разрядное целое число со знаком. Если определение _LARGE_FILES задано перед включением каких-либо заголовочных файлов, то активируется среда с поддержкой больших файлов, а тип данных **off_t** определяется как 64-разрядное целое число со знаком. Кроме того, вызовы всех процедур, в которых используются указатели смещения в файлах и размер файлов, будут заменены на вызовы аналогичных процедур, поддерживающих большие файлы. Соответственно, будут переопределены все структуры данных, содержащие поля размера файла и смещения в файле.

В следующей таблице указано, какие определения изменяются в среде _LARGE_FILES:

Объект	Заменяется на	Заголовочный файл
Объект off_t	long long	<sys/types.h>
Объект fpos_t	long long	<sys/types.h>
Структура struct stat	struct stat64	<sys/stat.h>
Функция stat	stat64()	<sys/stat.h>
Функция fstat	fstat64()	<sys/stat.h>
Функция lstat	lstat64()	<sys/stat.h>
Функция mmap	mmap64()	<sys/mman.h>
Функция lockf	lockf64()	<sys/lockf.h>
Структура struct flock	struct flock64	<sys/flock.h>
Функция open	open64()	<fcntl.h>
Функция creat	creat64()	<fcntl.h>
Параметр команды F_GETLK	F_GETLK64	<fcntl.h>
Параметр команды F_SETLK	F_SETLK64	<fcntl.h>
Параметр команды F_SETLKW	F_SETLKW64	<fcntl.h>
Функция ftw	ftw64()	<ftw.h>
Функция nftw	nftw64()	<ftw.h>
Функция fseeko	fseeko64()	<stdio.h>
Функция ftello	ftello64()	<stdio.h>
Функция fgetpos	fgetpos64()	<stdio.h>
Функция fsetpos	fsetpos64()	<stdio.h>
Функция fopen	fopen64()	<stdio.h>
Функция freopen	freopen64()	<stdio.h>
Функция lseek	lseek64()	<unistd.h>
Функция ftruncate	ftruncate64()	<unistd.h>
Функция truncate	truncate64()	<unistd.h>
Функция fclear	fclear64()	<unistd.h>
Функция pwrite	pwrite64()	<unistd.h>
Функция pread	pread64()	<unistd.h>
Структура struct aiocb	struct aiocb64	<sys/aio.h>
Функция aio_read	aio_read64()	<sys/aio.h>
Функция aio_write	aio_write64()	<sys/aio.h>
Функция aio_cancel	aio_cancel64()	<sys/aio.h>
Функция aio_suspend	aio_suspend64()	<sys/aio.h>
Функция aio_return	aio_return64()	<sys/aio.h>
Функция aio_error	aio_error64()	<sys/aio.h>
Структура liocb	liocb64	<sys/aio.h>
Структура lio_listio	lio_listio64()	<sys/aio.h>

Применение функций с поддержкой 64-разрядной файловой системы

Перекомпиляция программы в режиме _LARGE_FILES может быть нежелательной из-за большого числа неявных изменений, вносимых в программу при такой перекомпиляции. Если в программе очень мало функций, работающих с файловой системой, то целесообразно просто заменить имена этих функций. Ниже перечислены типы данных, структуры и процедуры 64-разрядной файловой системы:

```
<sys/types.h>
typedef long long off64_t;
typedef long long fpos64_t;
```

```
<fcntl.h>

extern int      open64(const char *, int, ...);
extern int      creat64(const char *, mode_t);

#define F_GETLK64
#define F_SETLK64
#define F_SETLKW64

<ftw.h>
extern int ftw64(const char *, int (*)(const char *,const struct stat64 *, int), int);
extern int nftw64(const char *, int (*)(const char *, const struct stat64 *, int,struct FTW *),int, int);

<stdio.h>

extern int      fgetpos64(FILE *, fpos64_t *);
extern FILE     *fopen64(const char *, const char *);
extern FILE     *freopen64(const char *, const char *, FILE *);
extern int      fseeko64(FILE *, off64_t, int);
extern int      fsetpos64(FILE *, fpos64_t *);
extern off64_t  ftello64(FILE *);

<unistd.h>

extern off64_t  lseek64(int, off64_t, int);
extern int      ftruncate64(int, off64_t);
extern int      truncate64(const char *, off64_t);
extern off64_t  fclear64(int, off64_t);
extern ssize_t  pread64(int, void *, size_t, off64_t);
extern ssize_t  pwrite64(int, const void *, size_t, off64_t);
extern int      fsync_range64(int, int, off64_t, off64_t);

<sys/flock.h>

struct flock64;

<sys/lockf.h>

extern int lockf64 (int, int, off64_t);

<sys/mman.h>

extern void     *mmap64(void *, size_t, int, int, int, off64_t);

<sys/stat.h>

struct stat64;

extern int      stat64(const char *, struct stat64 *);
extern int      fstat64(int, struct stat64 *);
extern int      lstat64(const char *, struct stat64 *);

<sys/aio.h>

struct aiocb64
int     aio_read64(int, struct aiocb64 *):
int     aio_write64(int, struct aiocb64 *);
int     aio_listio64(int, struct aiocb64 *[],
        int, struct      sigevent *);
int     aio_cancel64(int, struct aiocb64 *);
int     aio_suspend64(int, struct aiocb64 *[]);

struct liocb64
int     lio_listio64(int, struct liocb64 *[], int, void *);
```

Типичные ошибки при работе с большими файлами

При переносе программы в среду с поддержкой больших файлов может оказаться, что для работы программы требуется внести в нее определенные изменения. Большинство ошибок бывает связано с

неаккуратным составлением программы, что не заметно в среде с 32-разрядной структурой **off_t**, но приводит к сбоям в среде с 64-разрядной структурой **off_t**. В этом разделе описаны наиболее распространенные ошибки и способы их исправления.

Примечание: В следующих примерах предполагается, что **off_t** - это 64-разрядное смещение в файле.

Неправильный выбор типов данных

Самая распространенная ошибка в программах - выбор неправильного типа данных. Если программа хранит размеры файлов и положения указателей в переменных типа int, то значения этих величин могут быть усечены. Размеры файлов и смещение указателя в файле следует хранить в переменных типа **off_t**.

Неправильно:

```
int file_size;
struct stat s;

file_size = s.st_size;
```

Лучше:

```
off_t file_size;
struct stat s;
file_size = s.st_size;
```

Если функция принимает в качестве аргументов или возвращает 64-разрядные целочисленные значения, то эта функция и та функция, из которой она вызывается, должны *одинаково* интерпретировать типы аргументов и возвращаемого значения.

Если вместо 64-разрядного целого в функцию будет передано 32-разрядное целое, то не только этот, но и другие аргументы вызова функции могут быть проинтерпретированы неправильно, что сделает результат вызова функции непредсказуемым. Эта ошибка особенно серьезна при передаче скалярных значений в функции, рассчитанные на 64-разрядные целые числа.

Для того чтобы избежать подобных ошибок, следует правильно задавать прототипы функций. В следующих примерах **fexample()** - это функция, ожидающая в качестве параметра 64-разрядное смещение указателя в файле. В первом примере компилятор создаст вызов функции, в которую передается 32-разрядное целое число. Во втором примере явно указан модификатор типа "LL", и поэтому компилятор создаст правильный вызов функции. В последнем примере перед вызовом функции в программу помещен ее прототип, в котором указан тип аргумента. Это оптимальное решение, поскольку в этом случае сохраняется возможность переноса программы в 32-разрядную среду.

Неправильно:

```
fexample(0);
```

Лучше:

```
fexample(0LL);
```

Лучше всего:

```
\est:
```

Создание связей - информация для программистов

Связь - это соответствие между именем файла и i-узлом (жесткая связь) или между двумя именами файлов (символьная связь).

Связывание обеспечивает доступ к i-узлу по нескольким именам. В записях каталога имена файлов и i-узлы указываются попарно. Имена файлов упрощают работу пользователя с файлами, а i-узлы содержат

фактические дисковые адреса данных файла. Счетчик всех связей с i-узлом хранится в поле **i_nlink** i-узла. Функции создания и уничтожения связей работают с именами файлов, а не с дескрипторами файлов. Поэтому при создании связей файлы можно не открывать.

Процессы могут считывать и изменять содержимое i-узла под любым из связанных с ним имен файлов. В AIX различают жесткие и символьные связи.

Жесткие связи

Функция	Описание
link	Функция создания жесткой связи. Наличие жесткой связи гарантирует существование файла, так как при ее создании увеличивается на единицу значение счетчика связей **i_nlink** в i-узле.
unlink	Функция освобождения связи. Если освободить все жесткие связи с i-узлом, файл становится недоступным.

Жесткие связи должны связывать имена файлов с i-узлами, расположенными в той же файловой системе, так как номер i-узла является уникальным только в пределах одной файловой системы. Жесткая связь всегда устанавливается для конкретного файла, поскольку при создании жесткой связи создается запись каталога, в которой указывается соответствие имени нового файла и i-узла. Владельцем файла является создатель исходного файла, он же определяет права доступа к файлу. С другой стороны, все жесткие связи для операционной системы равноправны.

Пример: Если файл **/u/tom/bob** связан с файлом **/u/jack/foo**, то значение счетчика связей **i_nlink** файла **foo** будет равно 2. Обе жесткие связи равноправны. Если удалить **/u/jack/foo**, то он будет продолжать существовать под именем **/u/tom/bob** и будет доступен пользователям, обращающимся к каталогу **tom**. Однако владельцем файла останется `jack`, даже если файл **/u/jack/foo** удален. Пространство, выделенное для файла, считается пространством пользователя jack и учитывается при определении его лимита дискового пространства. Изменить принадлежность файла можно с помощью функции **chown**.

Символьная ссылка

Символьная связь представляет собой файл, содержащий путь. Для создания символьной связи служит команда **symlink**. Если процесс обнаруживает символьную связь, то содержащийся в ней путь добавляется к пути, по которому процесс выполняет поиск. Если в символьной связи указан полный путь, процесс выполняет поиск по дереву от корневого каталога до файла, имя которого указано в символьной связи. Если имя файла в символьной связи начинается не с символа / (косой черты), процесс предполагает, что этот путь отсчитывается от расположения символьной связи. Функция **unlink** удаляет как жесткие, так и символьные связи.

Символьные связи могут связывать файлы из разных файловых систем, так как операционная система считает их обычными файлами, а не частью структуры файловой системы. Наличие символьной связи не гарантирует существования целевого файла, так как символьная связь не влияет на поле **i_nlink** i-узла.

Функция	Описание
readlink	Функция чтения содержимого символьной связи. Многие функции (в том числе **open** и **stat**) работают с именами файлов, указанными в символьной связи.
lstat	Эта функция служит для создания отчета о состоянии файла, содержащего символьную связь, и не выполняет переход по указанной связи. Список процедур, выполняющих переход по символьной связи, приведен в описании функции **symlink**.

Символьные связи также называют *гибкими связями*, так как они ссылаются на файл по его имени. Если удалить или переместить целевой файл, то символьную связь невозможно будет установить.

Пример: Символьная связь с файлом **/u/joe/foo** - это файл, в котором хранится строка **/u/joe/foo**. Если владелец файла **foo** удалит его, то вызовы функций по символьной связи будут приводить к ошибке. Если

затем владелец создаст новый файл с именем **foo** в том же каталоге, то связь будет указывать на новый файл. Этот тип связи потому и называется гибкой связью, что i-узел, с которым она установлена, можно заменить другим.

В выводе команды **ls -l** символ `l` в первой позиции означает символьную связь. В последнем столбце вывода связи между файлами представлены в виде `Путь2 -> Путь1` (или `новое-имя -> старое-имя`).

Функция	Описание
unlink	Функция удаления записи каталога. Параметр *Путь* указывает имя отсоединяемого файла. По завершении вызова **unlink** счетчик связей i-узла уменьшается на 1.
remove	Эта функция также удаляет имя файла путем вызова функции **unlink** или **rmdir**.

Связи с каталогами

Функция	Описание
mkdir	Эта функция создает записи новых каталогов, создавая тем самым жесткие связи с с i-узлом, соответствующим новому каталогу.

Если нужно создать дополнительную связь с каталогом, рекомендуется создать символьную связь. Символьные связи не нарушают работу записей каталогов . и .. и поддерживают пустые стандартные каталоги. Ниже приведен пример пустого стандартного каталога **/u/joe/foo** и значений **i_nlink**.

`/u`

Значения	пустые значения	пустые значения	Каталог			
68			j	o	e	0

```
/u/joe
mkdir ("foo", 0666)
```

Значения	пустые значения	пустые значения	Каталог			
68			n	0	0	0
			n	n	0	0
235			f	o	o	0

`/u/joe/foo`

Значения	пустые значения	пустые значения	Каталог			
235			n	0	0	0
68			n	n	0	0

`Значений i_nlink`

i = 68

n_link 3

Для i = 68 n_link равно 3 (**/u; /u/joe; /u/joe/foo**).

i = 235

n_link 2

Для i = 235 n_link равно 2 (**/u/joe; /u/joe/foo**).

Работа с дескрипторами файлов

Дескриптор файла - это целое число без знака, с помощью которого процесс обращается к открытому файлу.

Количество дескрипторов файлов, доступных процессу, ограничено параметром **/OPEN_MAX**, заданным в файле **sys/limits.h**. Кроме того, количество дескрипторов файлов можно задать с помощью флага **-n** команды **ulimit**. Дескрипторы файлов создаются при выполнении функций **open**, **pipe**, **creat** и **fcntl**. Обычно каждый процесс работает с уникальным набором дескрипторов. Однако эти же дескрипторы могут применяться и дочерними процессами, созданными с помощью функции **fork**. Кроме того, дескрипторы можно скопировать с помощью функций **fcntl**, **dup** и **dup2**.

Дескрипторы файлов выполняют роль индексов таблицы дескрипторов, которая расположена в области **u_block** и создается ядром для каждого процесса. Чаще всего процесс получает дескрипторы с помощью операций **open** и **creat**, а также путем наследования от родительского процесса. При выполнении операции **fork** таблица дескрипторов копируется для дочернего процесса. В результате дочерний процесс получает право обращаться к файлам родительского процесса.

Таблицы дескрипторов файлов и системные таблицы открытых файлов

Структуры данных, содержащие список открытых файлов и список дескрипторов файлов, позволяют отслеживать обращения процессов к файлам и гарантировать целостность данных.

Таблица	Описание

Таблица дескрипторов файлов	Преобразует индексы таблицы (дескрипторы файлов) в указатели на открытые файлы. Для каждого процесса в области **u_block** создается своя собственная таблица дескрипторов. Каждая запись такой таблицы содержит следующие поля: поле флагов и указатель на файл. Допустимо не более **OPEN_MAX** дескрипторов файлов. Таблица дескрипторов файлов имеет следующую структуру:
	`struct ufd`
	`{`
	`  struct file *fp;`
	`  int flags;`
	`} *u_ufd`

Таблица открытых файлов	Содержит записи с информацией обо всех открытых файлах. В записи этой таблицы хранится текущее смещение указателя в файле, которое используется во всех операциях чтения и записи в файл, а также режим открытия файла (**O_RDONLY**, **O_WRONLY** или **O_RDWR**). В структуре таблицы открытых файлов хранится смещение указателя в файле. При выполнении операции чтения-записи система выполняет неявный сдвиг указателя. Например, при чтении или записи *x* байт указатель также будет перемещен на *x* байт. Для изменения положения указателя в файлах с прямым доступом применяется функция **lseek**. Для потоковых файлов (например, каналов и сокетов) понятие смещения не поддерживается, так как произвольный доступ к этим файлам невозможен.

Управление дескрипторами файлов

Поскольку с файлами может работать несколько пользователей, необходимо, чтобы связанные процессы работали с общим указателем смещения, а независимые процессы - с собственным указателем смещения в файле. В записи таблицы открытых файлов содержится счетчик обращений к файлу, отражающий число дескрипторов, соответствующих данному файлу.

Несколько обращений к файлу может потребоваться в следующих случаях:

- Файл открыт еще одним процессом
- Дочерний процесс унаследовал дескрипторы файлов, открытых родительским процессом
- Дескриптор файла скопирован с помощью функции **fcntl** или **dup**

Совместная работа с открытыми файлами

При выполнении каждой операции открытия в таблицу открытых файлов добавляется запись. Это гарантирует, что каждый процесс будет работать со своим указателем в файле. Такой подход позволяет сохранить целостность данных.

При копировании дескриптора два процесса начинают работать с одним и тем же указателем. В этом случае оба процесса могут попытаться одновременно обратиться к файлу, при этом данные будут считаны или записаны не последовательно.

Копирование дескрипторов файлов

Существуют следующие способы копирования дескрипторов файлов: функция **dup** или **dup2**, функция **fork** и функция **fcntl**.

Функции dup и dup2

Функция **dup** создает копию дескриптора файла. Копия создается в пустой строке пользовательской таблицы дескрипторов, содержащей исходный дескриптор. При вызове **dup** увеличивается значение счетчика обращений к файлу в записи таблицы открытых файлов и возвращается новый дескриптор файла.

Функция **dup2** находит запрошенный дескриптор и закрывает связанный с ним файл, если он открыт. С ее помощью можно указать конкретную запись таблицы, в которую должен быть скопирован дескриптор.

fork, функция

Функция **fork** создает дочерний процесс, который наследует все дескрипторы файлов родительского процесса. После этого дочерний процесс запускает новый процесс. Унаследованные дескрипторы с флагом **Закрыть при exec**, установленным с помощью **fcntl**, будут закрыты.

Функция fcntl

Функция **fcntl** позволяет работать со структурой данных о файле и с дескрипторами открытых файлов. Она позволяет выполнять следующие операции над дескрипторами:

- Копировать дескриптор файла (аналогично функции **dup**).
- Получать или устанавливать значение флага Закрыть при exec.
- Выключать режим объединения дескрипторов в блоки.
- Включать режим добавления данных в конец файла (**O_APPEND**).
- Включать отправку процессам сигнала о разрешении ввода-вывода.
- Устанавливать и получать ИД процесса или группы процессов для отправки **SIGIO**.
- Закрывать все дескрипторы файлов.

Стандартные дескрипторы файлов

При запуске программы в оболочке открывается три дескриптора 0, 1 и 2. По умолчанию с ними связаны следующие файлы:

Дескриптор	Описание
0	Стандартный ввод.
1	Стандартный вывод.
2	Стандартный вывод сообщений об ошибках.

Перечисленные дескрипторы файлов связаны с терминалом. Это означает, что при чтении данных из файла с дескриптором 0 программа получает ввод с терминала, а при записи данных в файлы с дескрипторами 1 и 2 они выводятся на терминал. При открытии других файлов дескрипторы присваиваются в порядке возрастания.

Если ввод-вывод перенаправляется с помощью операторов < (знак меньше) или > (знак больше), то стандартные дескрипторы связываются с другими файлами. Например, следующая команда связывает дескрипторы файлов 0 и 1 с необходимыми файлами (по умолчанию эти дескрипторы связаны с терминалом).

```
prog < FileX > FileY
```

В данном примере дескриптор 0 будет связан с файлом `FileX`, а дескриптор 1 - с файлом `FileY`. Дескриптор 2 не будет изменен. Программе достаточно знать, что дескриптор 0 представляет файл ввода, а дескрипторы 1 и 2 - файлы вывода. Информация о том, с какими конкретно файлами связаны эти дескрипторы, ей не нужна.

В следующем примере программы продемонстрировано перенаправление стандартного вывода:

```
#include <fcntl.h>
#include<stdio.h>

void redirect_stdout(char *);

main()
{
       printf("Hello world\n");       /* печать в стандартный
                                      вывод */
       fflush(stdout);
       redirect_stdout("foo");        /*перенаправление стандартного вывода*/
       printf("Hello to you too, foo\n");
                                      /*печать в файл foo */
       fflush(stdout);
}

void
redirect_stdout(char *filename)
{
        int fd;
        if ((fd = open(filename,O_CREAT|O_WRONLY,0666)) < 0)
                                       /*открытие нового файла*/
        {
                perror(filename);
                exit(1);
        }
        close(1);                      /*закрытие стандартного*/
                                       вывода */
        if (dup(fd) !=1)               /*присвоение новому дескриптору
                                       *значения 1*/
        {
                fprintf(stderr,"Unexpected dup failure\n");
                exit(1);
        }
        close(fd);                      /*закрытие ненужного*/
                                        * исходного fd*/
}
```

При получении запроса на дескриптор выделяется первый свободный дескриптор из таблицы дескрипторов (дескриптор с наименьшим номером). Однако с помощью функции **dup** файлу можно присвоить любой дескриптор.

Ограничение на число дескрипторов файлов

Максимальное число дескрипторов, которое может использоваться в одном процессе, ограничено. Значение по умолчанию указывается в файле **/etc/security/limits** и обычно равно **2000**. Для изменения ограничения можно воспользоваться командой **ulimit** или функцией **setrlimit**. Максимальное число определяется константой **OPEN_MAX**.

Создание и удаление файлов

В этом разделе описаны действия, которые выполняет операционная система при создании, открытии, закрытии или удалении файлов.

Создание файла

Для создания файлов различных типов предусмотрены разные функции:

Функция	Тип создаваемого файла
creat	Обычный
open	Обычный (если установлен флаг **O_CREAT**)
mknod	Обычный, FIFO или специальный
mkfifo	Именованный канал (FIFO)
pipe	Неименованный канал
socket	Сокеты
mkdir	Каталоги
symlink	Символьная связь

Создание обычного файла (функции creat, open и mknod)

Функция **creat** создает файл с указанными *именем* и *режимом доступа*. Если файл с таким *Именем* уже существует, и у процесса есть права на запись в него, то функция **creat** усекает файл. В результате освобождаются все блоки данных, а размер файла обнуляется. Обычный файл можно создать и с помощью функции **open** с флагом **O_CREAT**.

Функции **creat**, **mkfifo** и **mknod** создают файлы с правами доступа, заданными в параметре *Режим*. При создании обычного файла с помощью функции **open** права доступа задаются в параметре *Режим* флага **O_CREAT**. Функция **umask** устанавливает маску прав доступа, предоставляемых при создании файлов, и позволяет получить предыдущее значение маски.

Права доступа к новому файлу вычисляются путем побитового умножения величины, обратной **umask**, на права доступа, указанные процессом при создании файла. Во время создания файла процессом операционная система выполняет следующие действия:

- Определяет права доступа процесса, создающего файл.
- Получает значение **umask**.
- Вычисляет значение, обратное **umask**.
- Умножает права доступа на значение **umask**.

Создание особого файла (функции mknod и mkfifo)

Для создания специальных файлов применяются функции **mknod** и **mkfifo**. Функция **mknod** позволяет создать именованный канал (FIFO), обычный файл или файл устройства. Она создает такой же i-узел, как и функция **creat**. При создании файла с помощью функции **mknod** тип этого файла указывается в соответствующем параметре. При создании блочного или символьного файла устройства в i-узел записывается информация о главном и дополнительном устройствах.

Функция **mkfifo** служит интерфейсом для функции **mknod** и применяется для создания именованных каналов.

Открытие файла

Для обращения к ранее созданному файлу применяется функция **open**. Она возвращает дескриптор файла. Этот дескриптор применяется функциями чтения, записи, поиска, копирования, установки параметров ввода-вывода, определения состояния файла и закрытия файла. При выделении дескриптора файла функция **open** создает запись в таблице дескрипторов файлов.

Функция **open** выполняет следующие действия:

- Проверяет, есть ли у процесса права доступа к файлу.
- Выделяет для открытого файла запись в таблице дескрипторов файлов. Функция **open** устанавливает указатель чтения/записи в начало файла.

Функции **ioctl** и **ioctlx** предназначены для управления специальными файлами устройств.

Закрытие файла

Для удаления записи о ненужном файле из таблицы дескрипторов файлов применяется функция **close**. Если в таблице дескрипторов существует несколько ссылок на файл, то значение счетчика уменьшается на 1. Если существует только одна ссылка на файл, то запись таблицы освобождается. Обращение к файлу, дескриптор которого освобожден, приведет к ошибке, если только его дескриптор не был повторно связан с файлом с помощью функции **open**. При завершении процесса ядро закрывает все дескрипторы файлов, открытые и не освобожденные процессом. Это гарантирует, что к моменту завершения процесса все файлы будут закрыты.

Удаление файла

Если файл больше не нужен, его можно удалить с помощью функции **unlink**. Если на один и тот же файл указывает несколько жестких ссылок, функция **unlink** позволяет удалить указанную ссылку. Если существует только одна ссылка, функция **unlink** удаляет сам файл. Дополнительная информация приведена в описании функции **unlink**.

Работа с файлами

Во всех операциях ввода-вывода используется смещение указателя в файле, которое хранится в таблице открытых файлов.

Это смещение измеряется в байтах и отслеживается для каждого открытого файла. Смещение указателя в файле определяет позицию в файле, начиная с которой будут считываться или записываться данные. Функция **open** помещает указатель в начало файла. Изменить положение указателя можно с помощью функции **lseek**.

Изменение положения указателя в файле

Данные считываются из файла и записываются в файл последовательно, так как после выполнения каждой операции ввода-вывода смещение указателя в файле сохраняется. Оно хранится в таблице открытых файлов.

В файлах с прямым доступом, например, в обычных и особых файлах, положение указателя в файле можно изменить с помощью функции **lseek**.

Функция	Описание
lseek	Эта функция позволяет поместить указатель в заданную позицию файла. Позиция указателя в файле определяется переменной *Смещение*. Значение *Смещения* можно задавать относительно следующих позиций в файле (такая позиция указывается в переменной *Откуда*): **Абсолютное смещение** Смещение относительно начала файла **Относительное смещение** Смещение относительно прежнего положения указателя **Конец файла** Смещение относительно конца файла

Функция **lseek** возвращает текущее положение указателя в файле. Например:

```
cur_off= lseek(fd, 0, SEEK_CUR);
```

Результат выполнения **lseek** сохраняется в таблице открытых файлов. Все последующие операции чтения и записи будут выполняться с учетом нового положения указателя в файле.

Примечание: Для каналов и сокетов положение указателя изменять нельзя.

Функция	Описание
fclear	Эта функция создает в файле пустое пространство. Она обнуляет область файла, размер которой задается переменной *Число байт*, начиная с текущей позиции указателя в файле. Если при открытии файла был установлен флаг **O_DEFER**, то функция **fclear** будет недоступна.

Чтение файла

В этом разделе рассмотрены функции чтения.

Чтение файла

Функция	Описание
read	Эта функция копирует указанное число байт из открытого файла в заданный буфер. Копирование начинается с текущей позиции указателя в файле. Число байт и буфер указываются в параметрах *число-байт* и *буфер* соответственно.

Функция **read** выполняет следующие действия:

1. Проверяет правильность параметра *FileDescriptor* и наличие у процесса прав на **чтение**. После этого она обращается к записи таблицы открытых файлов, соответствующей данному *дескриптору файла*.
2. Устанавливает в файле флаг, указывающий, что выполняется операция чтения. Наличие такого флага гарантирует, что другие процессы не будет обращаться к файлу во время чтения.
3. Преобразует значение указателя в файле и значение параметра *число-байт* в адрес блока.
4. Копирует содержимое блока в буфер.
5. Копирует содержимое буфера в область памяти, на которую указывает переменная *буфер*.
6. Сдвигает указатель в файле на число прочитанных байт. Это гарантирует последовательное считывание данных.
7. Вычитает количество прочитанных байт из значения переменной *число-байт*.
8. Повторяет те же действия до тех пор, пока не будут прочитаны все данные.
9. Возвращает общее количество прочитанных байт.

Операция завершается, если файл пустой, прочитаны все запрошенные данные или произошла ошибка.

Для того чтобы избежать лишних итераций в блоке чтения, рекомендуется начинать чтение с начала блока данных и считывать целое количество блоков. Если процесс считывает несколько последовательных блоков, то операционная система предполагает, что дальше он также будет считывать данные по порядку.

I-узел блокируется на время операции чтения. Это означает, что во время чтения файла другие процессы не могут его изменять. Сразу после завершения чтения блокировка снимается. Если другой процесс изменит файл между двумя операциями чтения, то результат чтения будет разным, однако целостность данных будет сохранена.

В следующем примере показано, как с помощью функции read можно подсчитать число нулевых байтов в файле **foo**:

```
#include <fcntl.h>
#include <sys/param.h>

main()
{
        int fd;
        int nbytes;
        int nnulls;
```

```
        int i;
        char buf[PAGESIZE]; /* Удобный размер буфера */
        nnulls=0;
        if ((fd = open("foo",O_RDONLY)) < 0)
                exit();
        while ((nbytes = read(fd,buf,sizeof(buf))) > 0)
                for (i = 0; i < nbytes; i++)
                        if (buf[i] == '\0';
                                nnulls++;
        printf("%d nulls found\n", nnulls);
}
```

Запись данных в файл

В этом разделе рассмотрены функции записи.

Функция	Описание
запись	Эта функция считывает данные, объем которых задается переменной *число-байт*, из области памяти, на которую указывает переменная *буфер*, и добавляет их в файл с дескриптором *дескриптор-файла*. При этом выполняется примерно та же последовательность действий, что и для операции **read**. Текущее значение указателя в файле считывается из таблицы открытых файлов.

Если при записи данных в файле отсутствует блок, соответствующий текущему значению указателя, функция **write** запрашивает новый блок. Информация о нем заносится в описание i-узла, связанного с файлом. При добавлении блока фактически может быть добавлено несколько блоков, если они требуются файловой системе для адресации данных.

I-узел блокируется на время записи. Это означает, что во время записи другие процессы не могут изменять файл. Сразу после завершения записи файл разблокируется. Если другой процесс изменит файл между двумя операциями записи, то результат записи будет различным, однако целостность данных будет сохранена.

Функция **write** выполняет тот же цикл операций, что и функция **read**. За один цикл она записывает на диск один блок данных. Иногда требуется записать только часть блока. В этом случае функция **write** считывает блок с диска, чтобы сохранить данные, которые хранились в нем ранее. Если нужно записать целый блок данных, то старое содержимое блока не сохраняется, а заменяется целиком. Блоки последовательно записываются на диск до тех пор, пока объем записанной информации не станет равен указанному *числу-байт*.

Отложенная запись

Для включения режима отложенной записи нужно указать флаг **O_DEFER**. В этом режиме данные записываются на диск как временный файл. Процедура отложенной записи сохраняет данные в кэше, что позволяет быстрее обрабатывать последующие обращения к этим же данным. Механизм отложенной записи сокращает число обращений к диску. Многие программы, в том числе почтовые клиенты и текстовые редакторы, создают временные файлы в каталоге **/tmp** и удаляют их через непродолжительное время.

Если при открытии файла был указан флаг отложенной записи (**O_DEFER**), то данные не записываются в постоянную память до тех пор, пока не будет вызвана функция **fsync** или операция синхронной записи в файл (**write**) (в случае, если он был открыт с флагом **O_SYNC** flag). Функция **fsync** сохраняет на диске все изменения, внесенные в файл. Более подробная информация о флагах **O_DEFER** и **O_SYNC** приведена в описании функции **open**.

Усечение файлов

Функции **truncate** и **ftruncate** позволяют изменять длину обычных файлов. Для их выполнения необходимы права на запись в файл. Новый размер файла задается параметром *Длина*. При этом указанное число байт отсчитывается от начала файла, а не от текущей позиции указателя в файле. Если новая *длина* меньше

текущей длины файла, то усеченные данные удаляются. Если новая длина больше текущей, то дополнительное пространство заполняется нулями. Функция возвращает новое число блоков в файле и обновляет информацию о размере файла.

Сравнение прямого ввода-вывода и ввода-вывода с использованием кэша

Обычно JFS и JFS2 записывают страницы файла в кэш, расположенный в памяти ядра. При получении запроса на чтение файла JFS и JFS2 считывают данные с диска в кэш (если их там еще нет), а затем копируют из кэша в пользовательский буфер.

При получении запроса на запись данные копируются из пользовательского буфера в кэш. На диск данные записываются позже.

Такая стратегия кэширования может быть довольно эффективной при большом числе попаданий в кэш. Кроме того, она поддерживает алгоритмы упреждающего чтения и отложенной записи. Запись в файл выполняется в асинхронном режиме, что дает возможность приложению сразу продолжать работу, а не ждать выполнения запроса на ввод-вывод.

При выполнении операций прямого ввода-вывода пользовательский буфер обменивается данными напрямую с диском, минуя кэш. Операции прямого ввода-вывода для файлов аналогичны операциям прямого ввода-вывода для устройств. Приложения могут использовать операции прямого ввода-вывода при работе с файлами JFS и JFS2.

Преимущества прямого ввода-вывода

Основное преимущество прямого ввода-вывода состоит в том, что он позволяет снизить нагрузку на CPU за счет исключения промежуточных операций копирования из кэша в пользовательский буфер.

Такой механизм может применяться при небольшом числе попаданий в кэш. В этом случае системе все равно приходится обращаться к диску. Кроме того, прямой ввод-вывод рекомендуется использовать в приложениях с синхронной записью, так как данные будут записываться непосредственно на диск. В обоих случаях нагрузка на CPU будет снижена за счет исключения операций копирования в кэш.

Второе преимущество прямого ввода-вывода заключается в том, что он повышает эффективность кэширования остальных файлов. Каждый запрос на чтение и запись файла конкурирует в кэше с другими запросами. В результате данные других файлов могут быть вытеснены из кэша. Если к данным, добавленным в кэш, обращаются редко, то эффективность работы кэша снижается. Прямой ввод-вывод может применяться для тех файлов, кэширование которых не дает существенных преимуществ. При этом в кэше будут размещаться те файлы, для которых действительно имеет смысл применять стратегию кэширования.

Влияние прямого ввода-вывода на производительность

Хотя прямой ввод-вывод снижает нагрузку на CPU, общее время выполнения процесса обычно возрастает, особенно при выполнении небольших запросов. Такие издержки связаны с существенным различием между прямым вводом-выводом и операциями с участием кэша.

Чтение напрямую с диска

Чтение напрямую с диска выполняется синхронно, в отличие от обычной стратегии кэширования, когда данные копируются на диск из кэша. Такой способ чтения оказывается крайне неэффективным в тех случаях, когда данные находятся в оперативной памяти в соответствии со стратегией кэширования.

Прямой ввод-вывод не поддерживает алгоритм упреждающего чтения JFS и JFS2. Этот алгоритм позволяет значительно повысить производительность последовательных операций чтения файла за счет увеличения объема считываемых данных и выполнения чтения параллельно с работой приложения.

Для того чтобы компенсировать отсутствие алгоритма упреждающего чтения, рекомендуется запрашивать чтение большого объема данных. Для того чтобы производительность чтения напрямую с диска была сравнима с производительностью алгоритма упреждающего чтения файловой системы JFS или JFS2, объем одновременно считываемых данных должен быть не меньше 128 КБ.

Кроме того, алгоритм упреждающего чтения можно реализовать в самом приложении путем отправки асинхронных запросов для чтения данных с запасом напрямую с диска. Это можно сделать путем использования нескольких нитей, либо с помощью функции **aio_read**.

Запись напрямую на диск

Запись напрямую на диск выполняется синхронно, в отличие от обычной стратегии кэширования, когда данные копируются в кэш и лишь затем записываются на диск. Это отличие может привести к значительному снижению скорости работы приложений, в которых обычные операции ввода-вывода были заменены на прямой ввод-вывод.

Конфликт режимов доступа к файлу

Для того чтобы избежать конфликтов между программами, применяющими прямой ввод-вывод и ввод-вывод с использованием кэша, операции прямого ввода-вывод всегда выполняются только в исключительном режиме. Если файл открыт несколькими процессами, причем некоторые из них выполняют прямой ввод-вывод, а некоторые используют кэш, то сохраняется обычный режим чтения-записи с помощью кэша. Данные будут напрямую считываться и записываться на диск только в том случае, если все процессы выполняют операции прямого ввод-вывода.

Аналогично, файл останется в режиме ввода-вывода с применением кэша, если он размещен в виртуальной памяти с помощью вызова **shmat** или **mmap**.

В случае конфликта JFS или JFS2 попытается перевести файл в режим прямого ввода-вывода. Ввод-вывод с применением кэша при этом запрещается (с помощью функции **close**, **munmap** или **shmdt**). Переход в режим прямого ввода-вывода связан с дополнительными временными издержками, поскольку все страницы файла должны быть удалены из оперативной памяти, а измененные страницы - записаны на диск.

Включение режима прямого ввода-вывода

Для того чтобы включить режим прямого ввода-вывода для файла, нужно указать в функции open флаг O_DIRECT. Этот флаг определен в файле **fcntl.h**. Для того чтобы в приложении было доступно определение O_DIRECT, его нужно скомпилировать с опцией _ALL_SOURCE.

Требования к смещению, размеру и адресу целевого буфера

Для того чтобы запросы на прямой ввод-вывод обрабатывались эффективно, они должны удовлетворять некоторым условиям. Приложения могут получить информацию о требованиях к смещению, размеру и адресу с помощью функций **finfo** и **ffinfo**. При работе с командой FI_DIOCAP функции **finfo** и **ffinfo** возвращают структуру **diocapbuf**, описанную в файле **sys/finfo.h**. Эта структура содержит следующие поля:

dio_offset
Рекомендуемая граница для выравнивания смещения в случае прямой записи в файл

dio_max
Рекомендуемый максимальный объем данных, записываемых в файл напрямую

dio_min
Рекомендуемый минимальный объем данных, записываемых в файл напрямую

dio_align
Рекомендуемая граница выравнивания для буфера, применяемого в операциях прямой записи в файл

Отклонение от рекомендуемых значений может привести к тому, что чтение и запись будут выполняться в обычном режиме с кэшированием, а прямой ввод-вывод будет заблокирован. В разных файловых системах к этим значениям предъявляются различные требования, как показано в следующей таблице.

Формат файловой системы	dio_offset	dio_max	dio_min	dio_align
Фиксированный JFS, блоки по 4 КБ	4 КБ	2 МБ	4 КБ	4 КБ
Фрагментированный JFS	4 КБ	2 МБ	4 КБ	4 КБ
Сжатый JFS	н/д	н/д	н/д	н/д
Большой файл JFS	128 КБ	2 МБ	128 КБ	4 КБ
JFS2	4 КБ	4 ГБ	4 КБ	4 КБ

Ограничения прямого ввода-вывода

Прямой ввод-вывод не поддерживается для файлов, расположенных в файловых системах со сжатием данных. Если при открытии файла будет задан флаг O_DIRECT, то он будет проигнорирован. Файл будет открыт в обычном режиме ввода-вывода с кэшированием.

Прямой ввод-вывод и целостность данных

Несмотря на то, что запись напрямую на диск выполняется синхронно, она не обеспечивает целостность данных в той мере, которая требуется в стандарте POSIX. Если в приложении предъявляются строгие требования к целостности данных, при открытии файла вместе с флагом O_DIRECT нужно указать флаг O_DSYNC. O_DSYNC гарантирует, что в постоянную память будут записаны все данные, а также достаточный объем метаданных (например, косвенных блоков). Это означает, что в случае сбоя системы все данные можно будет восстановить. При указании флага O_DIRECT без флага O_DSYNC записываются только обычные данные, но не метаданные.

Работа с каналами

Канал - это неименованный объект, позволяющий процессам обмениваться данными.

При этом один из процессов записывает данные в канал, а другой их считывает. Такой тип файла называется также файлом FIFO. В файле FIFO блоки данных образуют очередь, в которой есть указатели чтения и записи (указывающие на голову и на хвост очереди), которые позволяют сохранять порядок элементов очереди. Максимальный размер элемента очереди в байтах задается системной переменной **PIPE_BUF**, которая определена в файле **limits.h**.

В оболочке неименованные каналы применяются для создания конвейеров команд. Большинство неименованных каналов создается именно оболочкой. Канал между процессами обозначается символом | (вертикальная черта). В следующем примере на экран выдается вывод команды **ls**:

```
ls | pr
```

По возможности все каналы рассматриваются как обычные файлы. Обычно положение указателя в файле хранится в таблице открытых файлов. Однако канал совместно используется двумя процессами, которые должны работать с одним и тем же файлом, но не с одним и тем же указателем. С другой стороны, при выполнении функции **open** в таблице открытых файлов создается запись, уникальная для процесса, а не для файла. Для решения этой проблемы в таблице открытых файлов создаются записи, общие для нескольких процессов.

Функции работы с каналами

Функция **pipe** создает канал между двумя процессами и возвращает два дескриптора файла. Дескриптор 0 открывается для чтения. Дескриптор 1 открывается для записи. Операция чтения получает данные в порядке их записи. Описанные дескрипторы файлов применяются функциями **read**, **write** и **close**.

В следующем примере дочерний процесс создает канал для связи с родительским процессом и отправляет по нему свой ИД:

```
#include <sys/types.h>
main()
{
        int p[2];
        char buf[80];
        pid_t pid;

        if (pipe(p))
        {
                    perror("ошибка канала");
             exit(1)'
        }
        if ((pid=fork()) == 0)
        {
                                    /* дочерний процесс */
             close(p[0]);           /*закрывает ненужный*/
                                    *дескриптор чтения*/
             sprintf(buf,"%d",getpid());
                                    /* создание данных */
                                    /*для отправки*/
             write(p[1],buf,strlen(buf)+1);
                     /*запись данных с учетом
                     /*байта null*/
             exit(0);
        }
                                     /*родительский процесс*/
        close(p[1]);                 /*закрыть ненужную сторону канала */
        read(p[0],buf,sizeof(buf));     /*чтение данных из канала*/
        printf("Сообщение дочернего процесса: %s/n", buf);
                                    /*вывод результата*/
        exit(0);
}
```

При попытке чтения из пустого канала операция будет отложена до появления данных в канале. При попытке записи в переполненный канал (**PIPE_BUF**), операция будет отложена до освобождения необходимого пространства в канале. Если к моменту чтения данных из канала дескриптор записи будет уже закрыт, то операция чтения вернет признак конца файла.

К прочим функциям, предназначенным для работы с каналом, относятся **popen** и **pclose**.

popen Создает канал (с помощью функции **pipe**) и копию процесса, из которого она была вызвана. Дочерний процесс закрывает ненужный дескриптор канала (в зависимости от того, будет ли он записывать или считывать данные) и вызывает оболочку для запуска требуемого процесса с помощью функции **execl**.

Родительский процесс закрывает ненужный дескриптор канала. Все ненужные дескрипторы должны быть закрыты для обеспечения правильной передачи признака конца файла. Например, если дочерний процесс, читающий данные из файла, не закроет дескриптор записи, то он никогда не получит признак конца файла, так как один процесс записи всегда будет потенциально активным.

Ниже описан удобный способ связывания дескриптора файла канала со стандартным вводом процесса:

```
close(p[1]);
close(0);
dup(p[0]);
close(p[0]);
```

Функция **close** освобождает дескриптор 0 - стандартный ввод. Функция **dup** возвращает копию ранее открытого дескриптора файла. Дескрипторы файлов выделяются в порядке возрастания их номеров, причем всегда возвращается первый свободный дескриптор. Следовательно, функция **dup** выделит для дескриптора чтения канала дескриптор 0. Таким образом, стандартный ввод будет связан с дескриптором чтения канала. Старый дескриптор чтения закрывается. Аналогичное действие должно выполняться дочерним процессом, который записывает данные в канал связи с родительским процессом.

pclose Закрывает канал между программой, отправившей запрос, и командой оболочки, которая должна быть выполнена. Функция **pclose** позволяет закрыть все потоки, открытые с помощью **popen**.

Функция **pclose** дожидается завершения процесса, закрывает канал и возвращает код завершения команды. Эта функция удобнее функции **close**, так как она позволяет дочернему процессу закончить работу, и только после этого закрывает канал. Кроме того, в процессе может быть только ограниченное число незавершенных дочерних процессов, даже с учетом тех процессов, которые фактически уже выполнили свою задачу. Функция pclose всегда дожидается завершения процесса, поэтому данное ограничение никогда не будет превышено.

Синхронный ввод-вывод

По умолчанию данные записываются в файлы JFS и JFS2 асинхронно.

Однако JFS и JFS2 поддерживают следующие типы синхронного ввода-вывода:

- Тип, задаваемый флагом O_DSYNC. Если он будет указан при открытии файла, системный вызов write () будет возвращать управление программе только после записи в постоянную память всех данных и метаданных, необходимых для восстановления этих данных.
- Тип, задаваемый флагом O_SYNC. При указании этого флага функция write() будет выполнять те же действия, что и для флага O_DSYNC. Кроме того, перед передачей управления программе она запишет в постоянную память все атрибуты файла, связанные с вводом-выводом, даже если они не нужны для восстановления файла.

  Перед появлением флага O_DSYNC его функции в AIX выполнял флаг O_SYNC. Этот тип ввода-вывода поддерживается для обеспечения двоичной совместимости. Для того чтобы воспользоваться фактическими функциями флага O_SYNC, при открытии файла нужно указать оба флага O_DSYNC и O_SYNC. Флаг O_SYNC также будет выполнять свои функции, если будет экспортирована переменная среды **XPG_SUS_ENV=ON**.
- Тип, задаваемый флагом O_RSYNC. Он выполняет функции флагов O_SYNC и _DSYNC для операций чтения. Для файлов JFS и JFS2 флаг O_RSYNC имеет смысл указывать только в сочетании с флагом O_SYNC. В этом случае системный вызов read будет возвращать управление программе только после записи в постоянную память времени обращения к файлу.

Состояние файла

Информация о состоянии файла хранится в i-узле.

Получить эту информацию можно с помощью функций **stat**. Функции **stat** возвращают следующую информацию о файле: тип, владелец, режим доступа, размера файла, число связей, номер i-узла и время доступа к файлу. Эти функции записывают информацию в структуру, обозначаемую переменной *Buffer*. У процесса должны быть права на поиск в каталогах, образующих путь к указанному файлу.

Функция	Описание
stat	Возвращает информацию о файлах, имена которых заданы в параметре *Path*. Если невозможно представить размер файла в структуре, обозначенной переменной *Buffer*, то функция **stat** завершает работу аварийно и возвращает errno = EOVERFLOW.
lstat	Возвращает информацию о символьной связи, причем информацию о файле, с которым установлена эта связь, возвращает функция **stat**.
fstat	Возвращает информацию об открытом файле с использованием дескриптора файла.

Функции **statfs**, **fstafs** и **ustat** выдают информацию о состоянии файловой системы.

Функция	Описание
fstatfs	Возвращает информацию о файловой системе, в которой находится файл, связанный с данным дескриптором. Описание структуры возвращаемой информации приведено в файле **/usr/include/sys/statfs.h** для функций **statfs** и **fstatfs** и в файле **ustat.h** для функции **ustat**.
statfs	Возвращает информацию о файловой системе, содержащей файл, указанный в параметре *Path*.
ustat	Возвращает информацию о смонтированной файловой системе, обозначенной переменной *Device*. Идентификатор устройства обозначает любой файл, и его значение может быть определено с помощью поля `st_dev` структуры **stat**, определенной в файле **/usr/include/sys/stat.h**. Функция **ustat** имеет более низкий приоритет, чем функции **statfs** и **fstatfs**.
utimes и **utime**	Изменяют время обращения и изменения файла в i-узле.

Права доступа к файлам

При создании каждого файла для него устанавливается набор режимов доступа. Каждый режим описывает права на чтение, запись и выполнение, предоставленные пользователям, группе и всем остальным пользователям.

Права доступа к новому файлу вычисляются путем побитового умножения величины, обратной **umask**, на значение режима доступа, указанное процессом при создании файла. Во время создания файла процессом операционная система выполняет следующие действия:

- Определяет права доступа процесса, создающего файл.
- Получает значение **umask**.
- Вычисляет значение, обратное **umask**.
- Умножает права доступа на значение **umask**.

Например, режим доступа 027 указывает, что у владельца нет никаких прав доступа к файлу. Группе предоставлены права на запись. Всем остальным пользователям предоставлены права на чтение, запись и выполнение. Значение **umask** для набора прав доступа 027 равно 750 (обратно исходным правам доступа). После побитового умножения 750 на 666 (режим доступа к файлу, указанный в системном вызове при создании файла) получаются фактические права доступа к файлу - 640. Ниже эти права доступа записаны в другом формате:

```
027 = _ _ _  _ W _  R W X        Существующий режим доступа к файлу
750 = R W X  R _ X  _ _ _        Обратное значение (umask)
                                 прав доступа
666 = R W _ R W _ R W _ Режим доступа при создании файла
ANDED TO
750 = R W X R _ X _ _ _ Значение umask
640 = R W _ R _ _ _ _ _ Режим доступа к файлу
```

Функция	Описание
функция **access**	Анализирует и выдает права доступа к файлу с указанным *путем*. Эта функция использует не действующие, а фактические ИД пользователя и группы. Это позволяет программам, изменившим свой ИД пользователя и группы, предоставлять доступ лишь отдельным пользователям.
Функции **chmod** и **fchmod**	Изменяют права доступа к файлу.
функция **chown**	Изменяет ИД владельца файла, указанный в i-узле. Старый ИД владельца удаляется, а на его место записывается новый. Функция **chmod** изменяет режим доступа к файлу.
umask	Получает и устанавливает маску создания файла.

В следующем примере у пользователя нет прав доступа к файлу `secrets`. Однако программа `special`, запущенная от имени `root`, сможет обратиться к файлу. Для того чтобы сохранить защиту файла, в программе должна применяться функция **access**.

```
$ ls -l
total 0
-r-s--x--x      1 root  system   8290 Jun 09 17:07 special
-rw-------      1 root  system   1833 Jun 09 17:07 secrets
$ cat secrets
cat: невозможно открыть secrets
```

Для того чтобы запретить доступ к файлу таким способом, в программах, изменяющих свои ИД пользователя и группы, всегда должна применяться функция **access**. При изменении владельца файла и режима доступа обновляется только i-узел, а не данные в файле. Эти действия может выполнять только процесс пользователя root или владельца файла.

Создание новых типов файловых систем

Для создания нового типа файловой системы необходимо разработать драйвер для самой файловой системы и для монтирования.

В этом разделе описаны особенности реализации и синтаксис вызова драйверов файловой системы и монтирования.

Драйверы файловых систем

В большинстве команд файловой системы нет специального кода для работы с файловыми системами конкретного типа. Такие команды собирают параметры, имена файловых систем и другую информацию, относящуюся к файловым системам всех типов, и передают ее базовой программе. Базовая программа предназначена для работы с файловой системой конкретного типа. Базовые программы, используемые командами файловой системы, называются *драйверами файловой системы* и *драйверами монтирования*.

При выполнении команды, обрабатывающей файлы, выполняется поиск драйвера с именем в формате **/sbin/helpers/*vfstype*/*command***, где *vfstype* - тип файловой системы, указанный в файле **/etc/vfs**, а *command* - имя выполняемой команды. Флаги команды передаются драйверу файловой системы.

Для каждого типа файловой системы должен быть предоставлен драйвер fstype, который не соответствует никакой команде. Он должен определять, действительно ли на указанном логическом томе находится файловая система соответствующего типа.

- Драйвер возвращает 0, если на логическом томе нет подходящей файловой системы. Значение 0 указывает на то, что логический том не содержит протокол.
- Драйвер возвращает 1, если на логическом томе есть файловая система нужного типа, и для протокола файловой системы не требуется отдельное устройство. Значение 1 указывает на то, что логический том содержит протокол.
- Драйвер возвращает значение 2, если на логическом томе есть подходящая файловая система, и для ее протокола требуется отдельное устройство. Если задан флаг **-l**, драйвер fstype должен проверить наличие протокола файловой системы на указанном логическом томе.

Устаревшая схема работы с файловыми системами

В этом разделе описана устаревшая схема работы с файловыми системами, применявшаяся в предыдущих версиях AIX. Ей по-прежнему можно пользоваться, хотя это и не рекомендуется.

Операции драйвера файловой системы

Ниже приведен список всех операций, которые может выполнять драйвер файловой системы (они описаны в файле **/usr/include/fshelp.h**):

Операции	Значение
#define FSHOP_NULL	0
#define FSHOP_CHECK	1
#define FSHOP_CHGSIZ	2
#define FSHOP_FINDATA	3
#define FSHOP_FREE	4
#define FSHOP_MAKE	5
#define FSHOP_REBUILD	6
#define FSHOP_STATFS	7
#define FSHOP_STAT	8
#define FSHOP_USAGE	9
#define FSHOP_NAMEI	10
#define FSHOP_DEBUG	11

Файловая система JFS поддерживает только следующие операции:

```
Операция Значение Команда

#define FSHOP_CHECK     1    fsck

#define FSHOP_CHGSIZ    2    chfs

#define FSHOP_MAKE      5    mkfs

#define FSHOP_STATFS    7    df

#define FSHOP_NAMEI     10   ff
```

Формат вызова драйвера файловой системы

Драйверы файловой системы имеют следующий формат вызова:

```
OpName OpKey FilsysFileDescriptor PipeFileDescriptor Modeflags
DebugLevel OpFlags
```

OpName
Задает параметр *arg0* при запуске драйвера из программы. Значение поля **OpName** помещается в список процессов (см. описание команды **ps**).

OpKey
Задает операцию драйвера. Например, если параметр **OpKey** равен 1, то выполняется операция **fsck** (проверить файловую систему).

FilsysFileDescriptor
Указывает дескриптор файла, по которому программа обратилась к файловой системе.

PipeFileDescriptor
Указывает файловый дескриптор конвейера (см. описание процедуры **pipe**), открытого между исходной программой и драйвером. Этот канал обеспечивает связь драйвера и вызывающей программы.

Пример: Драйвер передает по конвейеру сообщение об успешном выполнении операции или ошибке, в зависимости от чего определяется дальнейшая работа программы. В режиме отладки драйвер может передавать по конвейеру и другую информацию.

Modeflags

Описывает характер запуска драйвера и может влиять на его работу, особенно в отношении сообщений об ошибках. Флаги режимов определены в файле **/usr/include/fshelp.h**:

```
     Флаги                                Индикаторы
#define FSHMOD_INTERACT_FLAG                "i"
#define FSHMOD_FORCE_FLAG                   "f"
#define FSHMOD_NONBLOCK_FLAG                "n"
#define FSHMOD_PERROR_FLAG                  "p"
#define FSHMOD_ERRDUMP_FLAG                 "e"
#define FSHMOD_STANDALONE_FLAG              "s"
#define FSHMOD_IGNDEVTYPE_FLAG              "I"
```

Пример: Флаг **FSHMOD_INTERACT** указывает, будет ли команда выполняться интерактивно (определяется с помощью проверки значения функции **isatty** стандартного ввода). Не все операции поддерживают перечисленные выше режимы.

DebugLevel

Задает требуемый объем отладочной информации - чем выше уровень отладки, тем подробнее будет отладочная информация.

OpFlags

Указывает устройство (или устройства) для которого необходимо выполнить операцию, и все остальные опции, переданные вызывающей программой.

Примеры вызова драйверов

При запуске команды `fsck -fp /user` выполняется следующий вызов драйвера:

```
execl("/etc/helpers/v3fshelpers","fshop_check","1","3","5","ifp",
       "0","devices=/dev/lv02,fast,preen,mounted")
```

В этом примере:

- Команда **execd** запускает исполняемый файл `/etc/helper/v3fshelpers.`
- В списке процессов (команда **ps**) появляется имя `fshop_check`.
- Запрашивается операция **FSHOP_CHECK**, указанная значением `"1"`.
- Файловая система открывается для работы с дескриптором `"3"`.
- Канал, по которому драйвер может передавать информацию вызывающей программе, открыт с дескриптором `"5"`.
- Строке `ModeFlags` присвоено значение `"-ifp"`, что задает режимы `interactive`, `force` и `perror`.
- Значение `DebugLevel` равно `0`, поэтому дополнительная отладочная информация команде **fsck** передаваться не будет.
- Строка `OpFlags` указывает, для какого устройства запрошена операция (`/dev/lv02`), передает опции операции (`fast` и `preen`) и указывает, что устройство смонтировано. Для команды **fsck** это означает, что никакие изменения вноситься не будут, так как **fsck** не поддерживает смонтированные файловые системы.

Другой пример вызова драйверов рассматривается для команды **mkfs**. Для создания файловой системы JFS на существующем логическом томе **/dev/lv02** введите следующую команду создания точки монтирования:

```
mkfs /junk
```

Если при создании файловой системы известно только устройство, на котором ее следует смонтировать, введите:

```
mkfs /dev/lv02
```

В обоих случаях вызывается следующий драйвер файловой системы:

```
execl ("/etc/helpers/v3fshelpers", "fshop_make", "5", "3", "5", "-ip",\
  "0", "name=/junk,label=/junk,dev=/dev/lv02")
```

В этом примере запрошена операция **FSHOP_MAKE**. Указываются режимы `interactive` и `perror`. Строка `OpFlags` указывает точку монтирования и устройство.

Драйверы монтирования

Команда **mount** - это интерфейсная программа, запускающая драйвер для конкретных файловых систем. Драйверы команд **mount** и **umount** (или **unmount**) называются *драйверами монтирования*.

Как и другие команды файловой системы, команда **mount** считывает опции, заданные в командной строке, и интерпретирует их в контексте файловой системы, описанной файлом **/etc/filesystems**. С помощью информации из файла **/etc/filesystems**, команда вызывает драйвер монтирования, соответствующий типу заданной файловой системы. Например, если будет введена следующая команда **mount**, то она обратиться к файлу **/etc/filesystems** и считает раздел, описывающий файловую систему **/test**.

```
mount /test
```

На основании содержимого файла **/etc/filesystems** команда **mount** определит, что файловая система **/test** - это удаленная файловая система узла `host1`, смонтированная как NFS. Команда **mount** также учтет указанные опции монтирования.

Ниже приведен пример раздела файла **/etc/filesystems**:

```
/test:
        dev             = /export
        vfs = nfs
        nodename        = host1
        options  = ro,fg,hard,intr
```

По типу файловой системы (в данном случае это `nfs`) будет определен необходимый драйвер монтирования. Для этого команда сравнит тип файловой системы со значением первых полей записей файла **/etc/vfs**. В третьем поле строки с соответствующим первым полем будет указано имя драйвера.

Формат вызова драйвера монтирования

Ниже приведен пример вызова драйвера монтирования:

```
/etc/helpers/nfsmnthelp M 0 host1 /export /test ro,fg,hard,intr
```

В командах **mount** и **unmount** указывается шесть параметров. Первые четыре параметра совпадают для обеих команд:

`operation`
Указывает операцию, запрашиваемую у драйвера. Возможные значения - M (монтирование), Q (запрос) и U (размонтирование). Операция запроса устарела и не применяется.

`debuglevel`
Определяется численным значением флага **-D**. Ни команда **mount**, ни команда **unmount** не поддерживают флаг **-D**, поэтому его значение равняется 0.

`имя-узла`
Указывает имя узла при удаленном монтировании или содержит пустую строку при локальном монтировании. Команды **mount** и **unmount** не вызывают драйвер, если параметр `имя узла` пуст.

`объект`
Указывает имя локального или удаленного устройства, каталога или файла, для которого выполняется операция монтирования или размонтирования. Не все файловые системы поддерживают все возможные

комбинации. Например, удаленные файловые системы обычно не поддерживают монтирование устройств, в то время как локальные файловые системы не поддерживают остальные типы монтирования.

Ниже описаны остальные параметры команды **mount**:

точка монтирования
Определяет локальный каталог или файл, в котором будет смонтирован объект.

опции
Содержит список опций файловой системы, разделенных запятой. Значение этого параметра определяется полем **options** соответствующего раздела файла **/etc/filesystems** или флагом **-o** *Опции* командной строки (**mount -o** *Опции*). Команда **mount** также поддерживает флаг **-r** (только для чтения), и при формировании этого поля преобразует его в опцию `ro`.

Ниже описаны остальные параметры команды **unmount**:

vfsNumber
Задает уникальный номер размонтируемого объекта. Этот номер возвращается функцией **vmount** и может быть получен процедурами **mntctl** и **stat**. Драйверу это значение передается как первый параметр вызова процедуры **uvmount**, которая и выполняет размонтирование.

флаг
Определяет значение второго параметра процедуры **uvmount**. Значение 1 соответствует принудительному размонтированию, если указан флаг **-f** (**umount -f**). В противном случае этот параметр имеет значение 0. Не все файловые системы поддерживают принудительное размонтирование.

Программирование логических томов

Администратор логических томов (LVM) состоит из библиотеки функций LVM и драйвера логического тома. Их описание приведено ниже:

- Библиотека функций LVM. Эти функции предназначены для определения групп томов и обслуживания групп физических и логических томов.
- Драйвер логического тома. Он представляет собой драйвер псевдоустройства, обрабатывающий весь логический ввод-вывод. Он занимает промежуточное положение между файловой системой и драйверами жестких дисков. Драйвер логического тома преобразует логический адрес в физический, выполняет зеркальное отображение и перемещение поврежденных блоков, а затем направляет запрос на ввод или вывод драйверу физического диска. Для работы с драйвером логического тома предназначены функции **open**, **close**, **read**, **write** и **ioctl**.

Описание параметров **readx** и **writex**, а также операций **ioctl**, относящихся к драйверу логического тома, приведено в книге *Kernel Extensions and Device Support Programming Concepts*.

Дополнительная информация о логических томах приведена в *Управление операционной системой и устройствами*.

Библиотека функций для работы с логическими томами

Функции LVM предназначены для определения и обслуживания логических и физических томов из групп томов.

Эти функции применяются системными командами, предназначенными для работы с физическими и логическими томами. С помощью программного интерфейса библиотеки LVM можно создать альтернативные версии системных команд для работы с логическими томами или расширить их функции.

Примечание: Для получения и обновления структур данных ядра, описывающих группу томов, функции LVM используют системный вызов **sysconfig**, для применения которого необходимы права пользователя root. В связи с этим для работы с библиотекой функций LVM также необходимы права пользователя root.

Библиотека содержит следующие функции:

Службы	Описание
lvm_querylv	Выводит полную информацию о характеристиках логического тома.
lvm_querypv	Выводит полную информацию о характеристиках физического тома.
lvm_queryvg	Выводит полную информацию о характеристиках группы томов.
lvm_queryvgs	Выводит информацию о включенных группах томов системы.

операция J2_CFG_ASSIST ioctl

Операция **J2_CFG_ASSIST** ioctl возвращает статистику производительности файловой системы JFS2.

Операция **J2_CFG_ASSIST** ioctl возвращает структуру **cfg_assist**, определенную в файле **/usr/include/sys/lvdd.h**. Эта структура содержит следующие поля:

Поле	Описание
throughput	Средняя пропускная способность дисков в файловой системе, выраженная в Мб/с. Для поддерживаемых запоминающих устройств данные о пропускной способности передаются самим устройством; иначе возвращается пропускная способность файловой системы в среде выполнения.
latency	Средняя латентность всех дисков в файловой системе, выраженная в миллисекундах. Для поддерживаемых запоминающих устройств данные о пропускной способности передаются самим устройством; иначе возвращается латентность файловой системы в среде выполнения.
flags	Используемые флаги. Список допустимых флагов приведен в файле **/usr/include/sys/lvdd.h**.
vg_max_transfer	Максимальный передаваемый размер группы томов (VG), в Кб. Значение поля *vg_max_transfer* равно максимальному объему данных, которые могут быть переданы в одном запросе ввода-вывода на диски группы томов.
write_atomicity	Атомарность записи, в байтах. Значение поля *write_atomicity* равно максимальному числу байтов, не разбиваемых при записи в установленных границах.

Операция **J2_CFG_ASSIST** ioctl возвращает описанные ниже параметры только для поддерживаемых запоминающих устройств; иначе она возвращает нулевые значения.

Параметр	Описание
atomicWriteAlignment	Требуемое соответствие для атомарности записи в байтах.
ideal_sequential_read_size	Идеальный, последовательный, размер для чтения с дисков в файловой системе, в Кб.
ideal_sequential_write_size	Идеальный, последовательный, размер для записи на диски в файловой системе, в Кб.
ideal_random_read_size	Идеальный, произвольный, размер для чтения с дисков в файловой системе, в Кб.
ideal_random_write_size	Идеальный, произвольный, размер для записи на диски в файловой системе, в Кб.
stripsize	Размер полосы дисков в файловой системе, в Кб. Это объем данных, являющихся смежными на одном шпинделе в массиве raid.
stripesize	Значение параметра *Stripesize*, в Кб. (*Stripesize* = *stripsize* × число шпинделей в массиве RAID - проверка четности.
parallelism	Число шпинделей, из которых состоит устройство RAID, с которых возможно оперативное считывание или на которые возможна параллельная запись.

Коды возврата

По окончании выполнения этой операции возвращается значение 0. В случае сбоя операции, возвращается значение -1, а глобальная переменная **errno** получает одно из следующих значений:

Значение	Описание
EFAULT	Указывает на то, что копирование параметра не выполнено.
ENOMEM	Указывает на сбой при выделении памяти.
EAGAIN	Указывает на то, что статистика среды выполнения недоступна ни для одного из физических томов в файловой системе. Повторите попытку после выполнения дополнительного ввода-вывода в файловую систему.

Исключительные ситуации в операциях с плавающей точкой

В это разделе приведена информация об исключительных ситуациях, возникающих при выполнении операций с плавающей точкой, а также о способах отслеживания и обработки таких ситуаций.

Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт на исключительные ситуации операций с плавающей точкой, называемый Стандартом IEEE на двоичную арифметику с плавающей точкой (IEEE 754). Стандарт определяет пять типов исключительных ситуаций, которые должны отслеживаться:

- Недопустимая операция
- Деление на ноль
- Переполнение
- Потеря значимости
- Неточные исключительные ситуации

О возникновении любой из этих ситуаций сообщается путем установки флага или вызова прерывания. По умолчанию при возникновении исключительной ситуации система устанавливает флаг в регистрах состояния и управления операций с плавающей точкой (FPSCR). После установки эти флаги можно очистить только явным образом из процесса или при завершении процесса. В операционной системе предусмотрен набор функций для проверки, установки и очистки этих флагов.

Кроме того, при возникновении исключительной ситуации в операции с плавающей точкой система может отправить соответствующий сигнал (**SIGFPE**). Так как по умолчанию сигнал не отправляется, в операционной системе предусмотрены функции для изменения состояния процесса и разрешения этого сигнала. Если обработчик сигнала **SIGFPE** отсутствует, то при получении сигнала процесс завершается с созданием дампа. В противном случае вызывается функция-обработчик сигнала.

Функции для работы с исключительными ситуациями в операциях с плавающей точкой

Функции для работы с исключительными ситуациями в операциях с плавающей точкой позволяют выполнять следующие действия:

- Изменять состояние выполнения процесса
- Разрешить отправку сигнала при возникновении ошибки
- Отключать исключительные ситуации и очищать флаги
- Определять, какая исключительная ситуация была причиной сигнала
- Проверить флаги привязки

Для выполнения этих операций применяются следующие функции:

Функция	Процедура
fp_any_xcp и **fp_divbyzero**	Проверить флаги привязки
fp_enable и **fp_enable_all**	Разрешить отправку сигнала при возникновении ошибки
fp_inexact, **fp_invalid_op**, **fp_iop_convert**, **fp_iop_infdinf**, **fp_iop_infmzr**, **fp_iop_infsinf**, **fp_iop_invcmp**, **fp_iop_snan**, **fp_iop_sqrt**, **fp_iop_vxsoft**, **fp_iop_zrdzr** и **fp_overflow**	Проверить флаги привязки
fp_sh_info	Определить, какая исключительная ситуация была причиной сигнала
fp_sh_set_stat	Отключить исключительные ситуации или очистить флаги
fp_trap	Изменить состояние выполнения процесса
fp_underflow	Проверить флаги привязки
sigaction	Установить обработчик сигнала

Обработчик прерываний при ошибках в операциях с плавающей точкой

Для вызова прерываний программа должна изменить состояние выполнения процесса с помощью процедуры **fp_trap** и разрешить отправку сигнала с помощью функции **fp_enable** или **fp_enable_all**.

Изменение состояния выполнения процесса может снизить производительность, так как поддержка прерываний в операциях с плавающей точки требует перевода процесса в последовательный режим обработки.

При возникновении прерывания передается сигнал **SIGFPE**. По умолчанию при получении сигнала **SIGFPE** работа процесса завершается с созданием дампа памяти. В программе может быть определен обработчик сигнала, в котором будут выполняться определенные действия по обработке ошибки. Информация об обработчиках сигналов приведена в описании функций **sigaction**, **sigvec** и **signal**.

Исключительные ситуации: сравнение режима включенной и выключенной обработки

Ниже перечислены отличия двух режимов обработки и функций, используемых в этих режимах:

Модель с отключенной обработкой исключительных ситуаций

Следующие функции позволяют проверить флаги исключительных ситуаций в режиме с отключенной обработкой:

- **fp_any_xcp**
- **fp_clr_flag**
- **fp_divbyzero**
- **fp_inexact**
- **fp_invalid_op**
- **fp_iop_convert**
- **fp_iop_infdinf**
- **fp_iop_infmzr**
- **fp_iop_infsi**
- **fp_iop_invcmp**
- **fp_iop_snan**
- **fp_iop_sqrt**
- **fp_iop_vxsoft**
- **fp_iop_zrdzr**
- **fp_overflow**
- **fp_underflow**

Модель со включенной обработкой исключительных ситуаций

Следующие функции применяются в режиме со включенной обработкой исключительных ситуаций:

Функция	Состояние обработки
fp_enable и **fp_enable_all**	Разрешить отправку сигнала при возникновении ошибки
fp_sh_info	Определить, какая исключительная ситуация была причиной сигнала
fp_sh_set_stat	Отключить исключительные ситуации или очистить флаги
fp_trap	Изменить состояние выполнения процесса
sigaction	Установить обработчик сигнала

Режим неточных прерываний

В некоторых системах поддерживаются режимы *неточных прерываний*. Это означает, что аппаратное обеспечение обнаруживает исключительную ситуацию и вызывает прерывание, но работа программы во время передачи сигнала продолжается. В результате регистр адреса инструкции (IAR) указывает не на ту инструкцию, которая вызвала прерывание.

При работе в таком режиме снижение производительности менее заметно, чем при использовании *режима точных прерываний*. Однако в таком режиме ряд действий по исправлению ошибок становится невозможным, так как не всегда можно определить инструкцию, вызвавшую исключительную ситуацию; кроме того, последующие инструкции могут изменить аргумент, из-за которого произошла ошибка.

Для работы с неточными исключительными ситуациями обработчик сигналов должен определять, было ли прерывание точным или неточным.

Точные прерывания

В точных прерываниях регистр адреса инструкции (IAR) указывает на инструкцию, вызвавшую прерывание. Программа может изменить инструкцию и перезапустить ее или исправить результат и продолжить выполнение. Для продолжения работы необходимо увеличить IAR, чтобы он указывал на следующую инструкцию программы.

Неточные прерывания

При неточном прерывании IAR указывает на инструкцию, которая следует после инструкции, вызвавшей прерывание. Инструкция, на которую указывает IAR, еще не была выполнена. Для продолжения работы увеличивать значение IAR в обработчике сигнала не требуется.

Для устранения неоднозначности в структуре `trap_mode` предусмотрено поле **fp_sh_info**. В этом поле указывается режим прерываний для пользовательского процесса на момент входа в обработчик сигнала. Эта информация может быть также определена по регистру состояния компьютера (MSR) из структуры **mstsave**.

Функция **fp_sh_info** позволяет обработчику сигнала определить вид исключительной ситуации (точная или неточная).

> **Примечание:** Даже если включен режим точных прерываний, некоторые исключительные ситуации в операциях с плавающей точкой могут оставаться неточными (например, программно-реализованные операции). Аналогичным образом, в режиме неточных прерываний некоторые исключительные ситуации могут быть точными.

При использовании неточных исключительных ситуаций в некоторых частях программы для продолжения работы может потребоваться информация обо всех обнаруженных ошибках. Для этого применяется функция **fp_flush_imprecise** . Кроме того, рекомендуется использовать функцию **atexit** для регистрации функции **fp_flush_imprecise**, которая будет запускаться при выходе из приложения. Это позволит гарантировать, что по окончании работы программы необработанных исключительных ситуаций не останется.

Функции, зависящие от аппаратного обеспечения

В ряде систем функции вычисления квадратного корня числа с плавающей точкой и преобразования числа с плавающей точкой в целое число реализованы аппаратно. В других моделях для выполнения таких операций используются программные функции. При использовании любого метода может возникнуть ошибка, в результате которой будет вызвано прерывание. Если функция реализована программно, то она сообщает об ошибке с помощью функции **fp_sh_info**, указывая, что произошла неточная исключительная ситуация, так как IAR не указывает на инструкцию, вызвавшую ошибку.

Пример обработчика прерывания в операциях с плавающей точкой

```
/*
 * Ниже приведен пример обработчика прерываний для операций
 * с плавающей точкой.
 Обработчик идентифицирует исключительную
 * ситуацию и завершает свою работу.
 * Обработчик использует стандартный механизм возврата
 * сигналов - longjmp().
 */

#include <signal.h>
#include <setjmp.h>
#include <fpxcp.h>
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>

#define EXIT_BAD  -1
#define EXIT_GOOD  0

/*
 * Переменная согласования с обработчиком сигнала.
 * Если она равна нулю, применяется стандартный механизм
 * возврата из обработчика. При ненулевом значении
 * используется функция longjmp.
 */
static int fpsigexit;
#define SIGRETURN_EXIT 0
#define LONGJUMP_EXIT  1

static jmp_buf jump_buffer;      /* буфер для перехода */
#define JMP_DEFINED 0            /* код возврата setjmp при первом вызове */
#define JMP_FPE     2            /* код возврата setjmp при возврате */
                                 /* из обработчика сигнала */

/*
 * Структура fp_list позволяет связать
 * текстовое описание прерываний каждого типа
 * с маской, идентифицирующей тип.
 */

typedef struct
  {
  fpflag_t mask;
  char     *text;
  } fp_list_t;

/* типы прерываний, определенные IEEE */

fp_list_t
trap_list[] =
  {
      { FP_INVALID,      "FP_INVALID"},
      { FP_OVERFLOW,     "FP_OVERFLOW"},
      { FP_UNDERFLOW,    "FP_UNDERFLOW"},
      { FP_DIV_BY_ZERO,  "FP_DIV_BY_ZERO"},
      { FP_INEXACT,      "FP_INEXACT"}
  };

/* список вариантов INEXACT -- системное расширение стандартной обработки */
```

```
fp_list_t
detail_list[] =
  {
      { FP_INV_SNAN,   "FP_INV_SNAN" } ,
      { FP_INV_ISI,    "FP_INV_ISI" } ,
      { FP_INV_IDI,    "FP_INV_IDI" } ,
      { FP_INV_ZDZ,    "FP_INV_ZDZ" } ,
      { FP_INV_IMZ,    "FP_INV_IMZ" } ,
      { FP_INV_CMP,    "FP_INV_CMP" } ,
      { FP_INV_SQRT,   "FP_INV_SQRT" } ,
      { FP_INV_CVI,    "FP_INV_CVI" } ,
      { FP_INV_VXSOFT, "FP_INV_VXSOFT" }
  };
/*
 * Макроопределение TEST_IT применяется в функции main()
 * для вызова исключительной ситуации.
 */
#define TEST_IT(WHAT, RAISE_ARG)        \
  {                                     \
  puts(strcat("тест: ", WHAT));      \
  fp_clr_flag(FP_ALL_XCP);              \
  fp_raise_xcp(RAISE_ARG);              \
  }
/*
 * ИМЯ: my_div
 *
 * ФУНКЦИЯ:  Деление чисел с плавающей запятой.
 *
 */
double
my_div(double x, double y)
  {
  return x / y;
  }
/*
 * ИМЯ:      sigfpe_handler
 *
 * ФУНКЦИЯ:    Обработчик прерывания, которому передается управление при
 *             возникновении исключения операции с плавающей точкой. WBlazer примет
 *             точкой. Функция определяет, какой тип исключительной ситуации
 *             вызвал прерывание, выводит его в stdout и возвращает
 *             управление процессу, в котором произошло прерывание.
 *
 * ПРИМЕЧАНИЯ: Обработчик прерывания может завершаться,
 *             используя механизм по умолчанию, или с помощью longjmp().
 *             Метод определяется глобальной переменной fpsigexit.
 *
 *             При входе в функцию исключительные ситуации операций
 *             с плавающей точкой отключаются.
 *
 *             В примере применяется функция printf().
 *             Ее нужно использовать аккуратно, так как вывод
 *             этой функцией числа с плавающей точкой может вызвать ошибку.
 *             После этого вызов другого printf() для числа
 *             в обработчике исказит содержимое буфера,
 *             используемого при преобразовании.
 *
 * ВЫВОД:      Тип исключительной ситуации, вызвавшей
 *             прерывание.
 */
static void
sigfpe_handler(int sig,
               int code,
               struct sigcontext *SCP)
  {
```

```
  struct mstsave * state = &SCP->sc_jmpbuf.jmp_context;
  fp_sh_info_t flt_context;     /* структура для вызова
                                /* fp_sh_info() */
  int i;                        /* счетчик цикла */
  extern int fpsigexit;         /* глобальная переменная согласования */
  extern jmp_buf jump_buffer    /* буфер перехода */

  /*
   * Определить, какая исключительная ситуация вызвала прерывание.
   *             прерывание. * Функция fp_sh_info() применяется для получения структуры с
   * информацией о сигнале. После этого проверяется элемент структуры
   * flt_context.trap. Сначала тип прерывания сравнивается
   * со стандартными типами IEEE, и если прерывание вызвано
   * недопустимой операцией, проверяются дополнительные разряды.
   */

  fp_sh_info(SCP, &flt_context, FP_SH_INFO_SIZE);
static void
sigfpe_handler(int sig,
               int code,
               struct sigcontext *SCP)
  {
  struct mstsave * state = &SCP->sc_jmpbuf.jmp_context;
  fp_sh_info_t flt_context;     /* структура для вызова
                                /* fp_sh_info() */
  int i;                        /* счетчик цикла */
  extern int fpsigexit;         /* глобальная переменная согласования */
  extern jmp_buf jump_buffer    /* буфер перехода */

  /*
   * Определить, какая исключительная ситуация вызвала прерывание.
   *             прерывание. * Функция fp_sh_info() применяется для получения структуры с
   * информацией о сигнале. После этого проверяется элемент структуры
   * flt_context.trap. Сначала тип прерывания сравнивается
   * со стандартными типами IEEE, и если прерывание вызвано
   * недопустимой операцией, проверяются дополнительные разряды.
   */

  fp_sh_info(SCP, &flt_context, FP_SH_INFO_SIZE);
  for (i = 0; i < (sizeof(trap_list) / sizeof(fp_list_t)); i++)
      {
      if (flt_context.trap & trap_list[i].mask)
        (void) printf("Прерывание вызвано ошибкой %s \n", trap_list[i].text);
      }
  if (flt_context.trap & FP_INVALID)
      {
      for (i = 0; i < (sizeof(detail_list) / sizeof(fp_list_t)); i++)
          {
          if (flt_context.trap & detail_list[i].mask)
            (void) printf("Тип недопустимой операции - %s\n", detail_list[i].text);
          }
      }
  /* сообщить о текущем режиме прерываний */
  switch (flt_context.trap_mode)
      {
    case FP_TRAP_OFF:
      puts("Режим прерываний: OFF");
      break;

    case FP_TRAP_SYNC:
      puts("Режим прерываний: SYNC");
      break;

    case FP_TRAP_IMP:
      puts("Режим прерываний: IMP");
```

```
      break;

    case FP_TRAP_IMP_REC:
      puts("Режим прерываний: IMP_REC");
      break;

    default:
      puts("ОШИБКА:  Недопустимый режим прерываний");
      }
  if (fpsigexit == LONGJUMP_EXIT)
      {
      /*
       * Возврат по longjmp. В этой версии не требуется
       * очищать флаги исключительных ситуаций или отключать прерывания
       * для предотвращения зацикливания, так как при возврате из
       * обработчика процесс получит информацию о состоянии операций с
       * плавающей точкой обработчика сигнала.
       */
      longjmp(jump_buffer, JMP_FPE);
      }
  else
      {
      /*
       * Возврат с помощью стандартного механизма возврата из
       * обработчика сигнала. В этом случае требуется предупредить
       * зацикливание прерывания, или с помощью очистки разряда
       * исключительной ситуации в fpscr, или с помощью отключения прерываний.
       * В данном случае очищается разряд исключительной ситуации.
       * Для очистки применяется функция
       * fp_sh_set_stat.
       */
      fp_sh_set_stat(SCP, (flt_context.fpscr & ((fpstat_t) ~flt_context.trap)));
      /*
       * Увеличить iar процесса, вызвавшего прерывание
       * для предотвращения повторного выполнения инструкции.
       * Разряд FP_IAR_STAT в flt_context.flags определяет,
       * указывает ли state->iar на инструкцию, которая логически
       * запущена. Если этот бит равен 1, state->iar указывает на
       * операцию, которая была запущена и вызовет исключительную
       * ситуацию при повторном запуске. В данном случае требуется
       * продолжить выполнение, проигнорировав результаты операции
       * поэтому iar увеличивается таким образом, чтобы он указывал
       * на следующую инструкцию. Если этот разряд - нулевой, iar
       * уже указывает на следующую инструкцию.
       */

      if ( flt_context.flags & FP_IAR_STAT )
          {
          puts("Increment IAR");
          state->iar += 4;
          }
      }
  return;
  }

/*
 * ИМЯ:      main
 *
 * ФУНКЦИЯ:  Продемонстрировать работу обработчика sigfpe_handler.
 *
 */
int
main(void)
  {
  struct sigaction response;
  struct sigaction old_response;
```

```
extern int fpsigexit;
extern jmp_buf jump_buffer;
int jump_rc;
int trap_mode;
double arg1, arg2, r;

/*
 *  Установить обработку прерываний операций с плавающей точкой: Выполните следующие действия:
 *  1.  Очистить существующие флаги исключительных ситуаций.
 *  2.  Установить обработчик сигнала SIGFPE.
 *  3.  Перевести процесс в режим синхронного выполнения.
 *  4.  Разрешить все прерывания операций с плавающей точкой.
 */
fp_clr_flag(FP_ALL_XCP);
(void) sigaction(SIGFPE, NULL, &old_response);
(void) sigemptyset(&response.sa_mask);
response.sa_flags = FALSE;
response.sa_handler = (void (*)(int)) sigfpe_handler;
(void) sigaction(SIGFPE, &response, NULL);
fp_enable_all();
/*
 * Продемонстрировать обработчик со стандартным механизмом возврата.
 * Макроопределение TEST_IT вызывает исключительную ситуацию
 * указанного во втором аргументе типа. Проверка выполняется
 * в режиме точных прерываний, который поддерживается
 * всеми существующими платформами.
 */
trap_mode = fp_trap(FP_TRAP_SYNC);
if ((trap_mode == FP_TRAP_ERROR) ||
    (trap_mode == FP_TRAP_UNIMPL))
    {
    printf("ОШИБКА: fp_trap вернул код %d\n",
           trap_mode);
    exit(-1);
    }

(void) printf("Вызов обработчика по умолчанию: \n");
fpsigexit = SIGRETURN_EXIT;
TEST_IT("деление на ноль", FP_DIV_BY_ZERO);
TEST_IT("переполнение",    FP_OVERFLOW);
TEST_IT("потеря значимости",   FP_UNDERFLOW);
TEST_IT("неточное",     FP_INEXACT);
TEST_IT("сигнал NAN",      FP_INV_SNAN);
TEST_IT("INF - INF",          FP_INV_ISI);
TEST_IT("INF / INF",          FP_INV_IDI);
TEST_IT("0 / 0",        FP_INV_ZDZ);
TEST_IT("INF * 0",         FP_INV_IMZ);
TEST_IT("ошибка сравнения",    FP_INV_CMP);
TEST_IT("ошибка извлечения квадратного корня",       FP_INV_SQRT);
TEST_IT("ошибка преобразования",  FP_INV_CVI);
TEST_IT("программный запрос",   FP_INV_VXSOFT);
/*
 * Вызывать fp_trap() для определения
 * самого быстрого режима обработки прерываний
 * для текущей платформы.
 */
trap_mode = fp_trap(FP_TRAP_FASTMODE);
switch (trap_mode)
    {
  case FP_TRAP_SYNC:
    puts("Быстрый режимом для текущей платформы: PRECISE");
    break;
```

```
  case FP_TRAP_OFF:
    puts("Платформа не поддерживает прерывания");
    break;
  case FP_TRAP_IMP:
    puts("Быстрый режим для текущей платформы: IMPRECISE");
    break;
  case FP_TRAP_IMP_REC:
    puts("Быстрый режим для текущей платформы: IMPRECISE RECOVERABLE");
    break;
  default:
    printf("Непредвиденный код возврата fp_trap(FP_TRAP_FASTMODE): %d\n",
           trap_mode);
    exit(-2);
    }
/*
 * Если платформа поддерживает неточные прерывания, продемонстрировать режим.
 */

trap_mode = fp_trap(FP_TRAP_IMP);
if (trap_mode != FP_TRAP_UNIMPL)
    {
    puts("Демонстрация неточных исключительных ситуаций в операциях с плавающей точкой");
    arg1 = 1.2;
    arg2 = 0.0;
    r = my_div(arg1, arg2);
    fp_flush_imprecise();
    }
/* Продемонстрировать обработчик с возвратом по longjmp().
 */

(void) printf("возврат по longjump: \n");
fpsigexit = LONGJUMP_EXIT;
jump_rc = setjmp(jump_buffer);

switch (jump_rc)
    {
  case JMP_DEFINED:
    (void) printf("точка возврата установлена setjmp; идет проверка ...\n");
    TEST_IT("деление на ноль", FP_DIV_BY_ZERO);
    break;

  case JMP_FPE:
    (void) printf("Возврат из обработчика сигнала\n");
    /*
     * Учтите, что на текущем этапе состояние операций с плавающей точкой
     * процесса было получено от обработчика сигнала. Если в обработчике
     * не были включены прерывания (в этом примере они не включались),
     * то на этом этапе для процесса прерывания будут также отключены.
     * Мы создадим исключительную ситуацию, чтобы показать,
     * что прерывание не будет вызвано, а затем включим
     * прерывания.
     */
    (void) printf("Создается переполнение, прерывания не происходит\n");
    TEST_IT("Переполнение", FP_OVERFLOW);
    fp_enable_all();
    break;

  default:
    (void) printf("непредвиденный код возврата setjmp: %d\n", jump_rc);
    exit(EXIT_BAD);
    }
exit(EXIT_GOOD);
}
```

Информация, связанная с данной:

fp_clr_flag, fp_set_flag, fp_read_flag, fp_swag

fp_raise_xcp

sigaction, sigvec, snap Command

Управление вводом и выводом

Этот раздел содержит вводную информацию о программировании ввода-вывода и обзор функций управления вводом-выводом.

Функции библиотеки ввода-вывода предназначены для чтения данных из файлов (или получения от устройств) и записи данных в файл (или пересылки на устройство). Устройства рассматриваются системой как файлы ввода-вывода. Например, устройство необходимо открывать и закрывать точно так же, как и файл.

Некоторые функции используют для ввода-вывода данных стандартные файлы (устройства) ввода-вывода. Однако для большинства функций вы можете определять свои собственные файлы для ввода и вывода данных. Для многих функций можно использовать указатель на файл, т.е. на структуру, содержащую имя файла; для других вы можете использовать дескриптор файла (положительный целый идентификатор, присваиваемый файлу при его открытии).

Функции ввода-вывода, которые хранятся в библиотеке языка C (**libc.a**), выполняют потоковый ввод-вывод. Для доступа к этим функциям необходимо включить в программу файл **stdio.h** следующим образом:

```
#include <stdio.h>
```

Некоторые функции библиотеки ввода-вывода - это макрокоманды, определенные в файле заголовка, а некоторые - это объектные модули функций. Во многих случаях библиотека содержит макрокоманду и функцию, которые выполняют одну и ту же операцию. Выбирая между макрокомандой и функцией, учтите следующее:

- Для макрокоманды нельзя установить контрольную точку с помощью программы **dbx**.
- Макрокоманды обычно выполняются быстрее, чем функции, так как препроцессор заменяет макрокоманду на строки кода.
- При компиляции макрокоманд создается более громоздкий объектный код.
- При работе с функциями могут возникать побочные эффекты.

Файлы, команды и функции, применяемые для управления вводом-выводом, предоставляют следующие типы интерфейсов:

Низкий уровень
Низкоуровневый интерфейс предоставляет базовые функции открытия и закрытия файлов и устройств.

Поток
Потоковый интерфейс предоставляет функции чтения и записи для каналов и файлов FIFO.

Терминал
Интерфейс терминала обеспечивает форматированный вывод и буферизацию.

Асинхронный кабель
Асинхронный интерфейс обеспечивает одновременное выполнение операций ввода-вывода и обработки.

Язык ввода
Интерфейс языка ввода применяет команды **lex** и **yacc** для создания лексического анализатора, а также программу анализатора для интерпретации ввода-вывода.

Низкоуровневые интерфейсы ввода-вывода

Низкоуровневый интерфейс ввода-вывода - это точка непосредственного входа в ядро. Она предоставляет такие функции, как открытие и закрытие файлов, чтение из файла и запись в файл.

Для чтения одной строки из стандартного ввода предназначена команда **line**. Другие операции низкоуровневого ввода-вывода выполняются следующими функциями:

open, openx или creat
Подготовка файла или другого объекта каталога для чтения или записи путем присвоения ему дескриптора файла

read, readx, readv или readvx
Чтение данных из открытого файла с указанным дескриптором

write, writex, writev или writevx
Запись данных в открытый файл с указанным дескриптором

close
Освобождение дескриптора файла

Функции **open** и **creat** помещают записи в три системные таблицы. В первую таблицу, выполняющую функции области данных (доступ к которой имеют функции чтения и записи) для процесса, записываются индексы дескрипторов файлов. Каждая запись в этой таблице содержит указатель на соответствующую запись во второй таблице.

Вторая таблица - это база данных системы, или таблица файлов, которая позволяет нескольким процессам совместно использовать общие файлы. Запись в этой таблице указывает режим доступа к файлу (файл был открыт для чтения, для записи или как канал) или момент закрытия файла. Кроме того, запись содержит смещение, указывающее, где будет происходить следующая операция чтения или записи, и указатель на запись в третьей таблице, которая содержит копию индексного дескриптора (i-узла) файла.

В таблице файлов содержатся записи для всех экземпляров функции **open** или **create** в файле, а в таблицу i-узлов заносится только по одной записи для каждого файла.

Примечание: При обработке функций **open** или **creat** для конкретного файла система всегда вызывает функцию **open** для устройства, которая выполняет некоторые специальные операции (например, перематывает магнитную ленту или принимает сигнал DTR (сигнал готовности терминала) модема). Напротив, функция **close** используется системой только при закрытии файла последним процессом (т.е. при освобождении записи таблицы i-узлов файла). Это означает, что устройство может не зависеть от числа пользователей, за исключением случая, когда оно работает в монопольном режиме использования (в котором исключается его повторное открытие перед закрытием).

При выполнении операции чтения или записи для определения перечисленных ниже переменных используются аргументы, введенные пользователем, и информация, содержащаяся в записи таблицы файлов:

- Пользовательский адрес целевой области ввода-вывода
- Счетчик передаваемых байтов
- Текущее положение в файле

Если операции ввода-вывода выполняются над специальным текстовым файлом, то для передачи данных и обновления счетчика байтов и текущего положения указателя в файле вызывается соответствующая функция чтения или записи. В противном случае, текущее положение используется для вычисления номера логического блока в файле.

Если обрабатывается обычный файл, то номер логического блока должен быть преобразован в номер физического блока. Преобразовывать специальный блочный файл не нужно. Полученный номер физического блока используется для чтения данных с устройства или их записи.

Блочные драйверы устройств позволяют передавать информацию без использования буфера, непосредственно от пользовательского исполняемого модуля к устройству и обратно, причем размер буфера определяется в запросе вызывающей программы. Этот метод включает определение специального текстового файла, соответствующего устройству прямого доступа, и вызов функций чтения-записи для создания частного (не общего) заголовка буфера с соответствующей информацией. При необходимости могут вызываться отдельные функции открытия и закрытия, а также особая функция для работы с магнитной лентой.

Потоковые интерфейсы ввода-вывода

Потоковые интерфейсы, представляющие данные в виде потоков байтов, не интерпретируемых системой, более эффективны с точки зрения сетевых протоколов, чем символьные интерфейсы. При чтении и записи потока данных не происходит его деление на отдельные записи. Например, процесс, прочитавший 100 байт из некоторого канала, не может определить, были ли эти данные записаны сразу, либо было выполнено две операции по 50 байт, или же эти данные были записаны двумя различными процессами.

Потоковый ввода-вывод может быть организован в виде каналов или файлов FIFO ("первым вошел - первым вышел"). Файлы FIFO схожи с каналами, поскольку данные в них могут двигаться только в одном направлении (слева направо). Однако, в отличие от канала, файлу FIFO можно присваивать имя и к нему могут обращаться не связанные с ним процессы. Иногда файлы FIFO называют *именованными каналами*. Поскольку у файла FIFO есть имя, его можно открывать с помощью стандартной функции **fopen**. Процедура открытия канала сложнее: необходимо вызвать функцию **pipe**, возвращающую дескриптор файла, а затем с помощью стандартной функции ввода-вывода **fdopen** связать дескриптор открытого файла со стандартным потоком ввода-вывода.

Примечание: Потоковые интерфейсы ввода-вывода буферизуют данные на уровне пользователя и не могут записывать данные до тех пор, пока не будет выполнена процедура **fclose** или **fflush**, что может привести к непредвиденным результатам в случае смешивания с файловым вводом-выводом, например, read() или write().

Доступ к потоковым интерфейсам ввода-вывода можно получить с помощью следующих функций и макрокоманд:

fclose
Закрывает поток

feof, ferror, clearerr или fileno
Проверяет состояние потока

fflush
Записывает все буферизованные на данный момент символы из потока

fopen, freopen или fdopen
Открывает поток

fread или fwrite
Выполняет двоичный ввод

fseek, rewind, ftell, fgetpos или fsetpos
Перемещает указатель файла в потоке

getc, fgetc, getchar или getw
Извлекает символ или слово из входного потока

gets или fgets
Получает строку из потока

getwc, fgetwc или getwchar
Извлекает "широкий" символ из входного потока

getws или fgetws
Получает строку из потока

printf, fprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf или vwsprintf
Печатает форматированный вывод

putc, putchar, fputc или putw
Помещает символ или слово в поток

puts или fputs
Записывает строку в проток

putwc, putwchar или fputwc
Помещает символ или слово в поток

putws или fputws
Помещает строку "широких" символов в поток

scanf, fscanf, sscanf или wsscanf
Преобразует форматированный ввод

setbuf, setvbuf, setbuffer или setlinebuf
Выделяет буфер для потока

ungetc или ungetwc
Возвращает извлеченный символ обратно во входной поток

Терминальные интерфейсы ввода-вывода

Интерфейсы терминального ввода-вывода осуществляют взаимодействие между процессом и ядром, выполняя такие функции, как буферизация и форматирование вывода. Для каждого терминала или псевдотерминала существует структура tty, содержащая ИД текущей группы процесса. Это поле определяет группу процессов, получающих сигналы, связанные с данным терминалом. Доступ к терминальным интерфейсам ввода-вывода можно получить с помощью команды **iostat**, контролирующей загрузку системного устройства ввода-вывода, и демона **uprintfd**, позволяющего выводить сообщения ядра на экран терминала.

Ниже перечислены функции, позволяющие включать или отключать определенные параметры терминала:

cfgetospeed,cfsetospeed, cfgetispeed или cfsetispeed
Считывает и устанавливает значение скорости передачи в бодах

ioctl Выполняет функции управления терминалом

termdef
Запрашивает характеристики терминала

tcdrain
Ожидает завершения вывода

tcflow Выполняет функции управления потоком

tcflush Очищает указанную очередь

tcgetpgrp
Возвращает идентификатор интерактивной группы процессов

tcsendbreak
Передает сигнал прерывания по асинхронной последовательной линии

tcsetattr
Задает состояние терминала

ttylock, ttywait, ttyunlock или ttylocked
Управляет функциями блокировки терминала

ttyname и isatty
Получает имя терминала

ttyslot Выполняет поиск участка в файле **utmp** для текущего пользователя

Асинхронные интерфейсы ввода-вывода

Функции асинхронного ввода-вывода позволяют процессу запускать операцию ввода-вывода и сразу после запуска операции или постановки в очередь выходить из функции. Если необходимо, чтобы процесс ожидал завершения операции ввода-вывода (или немедленно получал управление обратно, если операция уже завершена), то требуется другая функция. Таким образом, в режиме асинхронного ввода-вывода процесс может совмещать операции обработки и ввода-вывода или выполнять ввод-вывод на несколько устройств одновременно. Применение режима асинхронного ввода-вывода практически не влияет на быстродействие процесса, который считывает данные из файла на диске и записывает их в другой файл на диске, но значительно повышает производительность других типов программ ввода-вывода, например, программ копирования содержимого диска на магнитную ленту или создания изображения на графическом дисплее.

Можно установить режим, при котором ядро будет уведомлять процесс, выполняющий асинхронный ввод-вывод, о готовности определенного дескриптора к вводу-выводу (так называемый *ввод-вывод по сигналу*). При работе с LEGACY AIO, для уведомления пользовательского процесса ядро использует сигнал SIGIO. При работе с POSIX AIO, для уведомления пользовательского процесса ядро использует структуру **sigevent**. Применяются следующие сигналы: **SIGIO**, **SIGUSR1** и **SIGUSR2**.

Для работы в режиме асинхронного ввода-вывода процесс должен выполнить следующие действия:

1. Установить обработчик сигнала **SIGIO**. Этот шаг необходим только в том случае, если установлен режим уведомления с помощью сигнала.
2. Установить ИД процесса или ИД группы процессов, которые будут принимать сигналы **SIGIO**. Этот шаг необходим только в том случае, если установлен режим уведомления с помощью сигнала.
3. Включить режим асинхронного ввода-вывода. Обычно решение о том, включать ли (загружать ли) этот режим, принимает системный администратор. Включение режима происходит при запуске системы.

Предусмотрены следующие функции асинхронного ввода-вывода:

`aio_cancel`
Отменяет один или несколько ожидающих запросов асинхронного ввода-вывода

`aio_error`
Получает информацию о состоянии ошибки запроса асинхронного ввода-вывода

`aio_fsync`
Синхронизирует асинхронные файлы.

`aio_nwait`
Приостанавливает вызывающий процесс до выполнения определенного числа запросов ввода-вывода.

`aio_read`
Считывает информацию в асинхронном режиме из файла с указанным дескриптором

`aio_return`
Определяет код возврата запроса асинхронного ввода-вывода

`aio_suspend`
Переводит вызывающий процесс в состояние ожидания, пока не завершится один или несколько запросов асинхронного ввода-вывода

`aio_write`
Записывает информацию в асинхронном режиме в файл с указанным дескриптором

`lio_listio`
Инициализирует список запросов асинхронного ввода-вывода посредством одного вызова

poll и select
Проверяет состояние ввода-вывода нескольких дескрипторов файлов и очередей сообщений

Для работы с функцией **poll** необходимо включить в программу следующие заголовочные файлы:

poll.h
Определяет структуры и флаги, используемые функцией **poll**

aio.h
Определяет структуры и флаги, используемые функциями **aio_read**, **aio_write** и **aio_suspend**

Ключи защиты памяти

Ключи защиты памяти обеспечивают механизм повышения надежности программ.

Ключи защиты применяются к страницам памяти и работают на уровне дискретности страниц, аналогично команде **mprotect**, которую можно использовать для для защиты от чтения или записи одной или нескольких страниц. Однако с помощью ключей памяти можно пометить разделы данных для конкретных уровней защиты от чтения и записи. Защита с помощью ключей памяти является функцией не только страницы данных, но также и функцией попытки доступа к нити. Можно включить определенные в исходном коде пути для доступа к данным, недоступные для более крупной программы, таким образом инкапсулируя важные данные программы и защищая их от случайных повреждений.

Поскольку доступ к защищенным ключами страницам является атрибутом выполняемой нити, этот механизм естественным образом распространяется на приложения с несколькими нитями, но с ограничением на использование этими приложениями только нитей pthread 1:1 (или области системы). Подход, реализуемый командой **mprotect**, не обеспечивает надежную работу в среде с несколькими нитями, поскольку приходится удалять защиту для всех нитей, если необходимо предоставить доступ для какой-либо одной нити. Можно одновременно использовать оба механизма; в этом случае, программа не может выполнить запись в защищенную от записи страницу даже если ключ защиты разрешает запись.

Пример использования ключей защиты включает следующее:

- Выполняется инкапсуляция всех частных данных программы, ограничивая доступ только выбранными путями исходного кода.
- Выполняется защита частных данных программы от случайного повреждения. При этом программа всегда выполняется с предоставленными правами на чтение, однако права на запись предоставляются только в том случае, если требуется изменить данные. Это может быть особенно полезным в том случае, если код в базовом механизме позволяет вызовы незащищенного кода.
- Если доступны несколько ключей защиты, то возможен дополнительный уровень дискретности при защите данных.

Можно упростить процесс отладки, если разрабатывать приложение, учитывая защиту ключами. Установка ключа защиты страницы и задание набора ключей активного пользователя являются системными вызовами, и поэтому являются относительно затратными операциями. Следует разрабатывать программу таким образом, чтобы частота этих операций не была избыточной.

Пользовательские ключи защиты

При работе с ключами защиты применяются следующие принципы и соглашения:

- Страницы, экспортированные из ядра только для чтения, по-прежнему будут видны для вашей программы. Эти страницы имеют ключ защиты **UKEY_SYSTEM**. Этот ключ защиты не находится под управлением программы, однако программа всегда имеет к нему доступ.
- Все страницы памяти вашей программы изначально имеют присвоенный им пользовательский общий ключ. Как было указано выше, доступ к памяти с ключом 0 предоставляется всегда, что делает этот ключ *пользовательским общим ключом*.

- Можно задать ключи защиты только для обычных и общих данных. Нельзя защитить, например, данные библиотеки, общую с ядром низшую память или текст программы.
- В зависимости от базового аппаратного обеспечения и административного выбора, доступно только ограниченное число пользовательских ключей защиты (обычно один). Когда программа присваивает ключ защиты одной или нескольким страницам, по умолчанию данные на этих страницах становятся недоступными. Необходимо явным образом предоставить права доступа к этим данным на чтение или запись для окружающих кодовых путей, которым требуется доступ, путем вызова новой службы для управления действующим пользовательским набором ключей.
- Физическое аппаратное обеспечение, вероятно, поддерживает дополнительные ключи защиты, которые не доступны для использования в качестве пользовательских ключей защиты.
- Для присвоения странице ключей защиты не требуются специальные права доступа. Единственным требованием является наличие текущих прав на запись для этой страницы.
- Ключи защиты не управляют правами на выполнение.

Если программа обращается к защищенным ключом данным в нарушение прав доступа, определенных в действующем наборе пользовательских ключей, она получит сигнал **SIGSEGV**, как в случае несанкционированного доступа к защищенным от чтения или записи страницам. Если вы указали, что этот сигнал следует обрабатывать, следует иметь в виду, что обработчики сигналов вызываются без доступа к частным ключам. Обрабатывающий сигнал код должен добавлять все необходимые права доступа к текущему пользовательскому набору ключей до обращения к защищенным ключом данным.

Дочерние процессы, созданные системным вызовом **fork**, логически наследуют состояние памяти и выполнения родительского процесса. Сюда входят ключи защиты, связанные с каждой страницей, а также действующий пользовательский набор ключей родительской нити во время вызова fork.

Области, защищенные пользовательскими ключами

Пользовательские ключи защиты могут использоваться для защиты страниц в следующих областях:
- Область данных
- Область стека по умолчанию
- Области **mmap**
- Общая память, присоединенная с помощью команды **shmat()**, за исключением перечисленных ниже случаев
- Для следующих категорий страниц нельзя использовать ключи защиты:
  - Файлы **shmat** и закрепленная общая память
  - Большие (нелистаемые) страницы
  - Текст программы
  - Общая с ядром нижняя область память с разрешением чтения

Системные предварительные требования для защиты ключами

Защита памяти ключами представляет собой зависящий от аппаратного обеспечения механизм защиты страниц, который предоставляется ядром AIX для применения в прикладных программах. Для использования этой функции к системе предъявляются следующие требования:
- Система должна выполняться на физическом аппаратном обеспечении, предоставляющем защиту памяти ключами
- Система должна выполняться в 64-разрядном ядре
- В системе должно быть включено применение пользовательских ключей защиты

Предварительные требования к программе для защиты ключами

Для использования пользовательских ключей к программе предъявляются следующие требования:

- Программа должна объявить себя поддерживающей пользовательские ключи и с помощью команды **ukey_enable** определить, сколько пользовательских ключей защиты доступно.
- Программа должна организовать защищенные данные в пределах страницы.
- Программа должна с помощью команды **ukey_protect** присвоить частный ключ для каждой страницы, которую необходимо защитить.
- Если данные malloc'd защищены, то перед высвобождением необходимо снять защиту.
- Программа должна с помощью команды **ukeyset_init** подготовить один или несколько наборов ключей.
- Программа, возможно, должна добавить необходимые ключи в набор ключей с помощью команды **ukeyset_add_key**, для включения в будущем прав на чтение или запись.
- Программа должна с помощью команды **ukeyset_activate** активировать набор ключей, чтобы предоставить права доступа, определенные в наборе.

Программа не должна:

- Включать какие либо нити pthread M:N (области процесса)
- Иметь возможность выполнения контрольной точки (например, при условии CHECKPOINT=yes в среде)

Примечание: Если программа поддерживает пользовательские ключи, существует связанный с ней дополнительный контекст для представления действующего пользовательского набора ключей. Это отображается в следующем:

- Обработчики сигналов, получающие структуру **ucontext_t**. Предыдущий действующий пользовательский набор ключей находится в **ucontext_t.__extctx.__ukeys**, массиве из двух элементов, содержащем 64-разрядное значение пользовательского набора ключей
- Пользовательские контекстные структуры, откомпилированные с определенным **__EXTABI__** (которое применяется **setcontext**, **getcontext**, **makecontext**, **swapcontext**)

Функции

Для работы с ключами защиты предоставляются следующие новые функции ядра AIX:

Функция	Описание
sysconf	**_SC_AIX_UKEYS** позволяет определить число поддерживаемых пользовательских ключей (эта функция может быть вызвана в предыдущих версиях AIX)
ukey_enable	Включает для процесса программную среду, поддерживающую пользовательские ключи, и сообщает, сколько пользовательских ключей доступно
ukeyset_init	Инициализирует пользовательский набор ключей, который будет представлять набор прав доступа для частного ключа (ключей)
ukeyset_add_key	Добавляет в набор ключей права на чтение или запись для указанного ключа
ukeyset_remove_key	Удаляет из набора ключей права на чтение и/или запись для указанного ключа
ukeyset_add_set	Добавляет все права доступа из одного набора ключей в другой
ukeyset_remove_set	Удаляет все права доступа одного набора ключей из другого
ukeyset_activate	Применяет права доступа набора ключей к выполняемой нити
ukeyset_ismember	Проверяет, содержатся ли данные права доступа в наборе ключей
ukey_setjmp	Расширенная форма **setjmp**, которая сохраняет действующий набор ключей (использует структуру **ukey_jmp_buf**)
pthread_attr_getukeyset_np	Получает атрибут набора ключей нити pthread
pthread_attr_setukeyset_np	задает атрибут набора ключей нити pthread
ukey_protect	Задает пользовательский ключ защиты для страничного диапазона пользовательской памяти
ukey_getkey	Получает пользовательский ключ защиты для указанного адреса

Отладка

Команда **dbx** добавляет ограниченную поддержку ключей защиты:

- При отладке выполняемой программы:
  - Команда **ukeyset** отображает действующий набор ключей.
  - Команда **ukeyvalue** отображает ключ защиты, связанный с данным расположением памяти.
- При отладке файла дампа команда **ukeyexcept** сообщает действующий набор ключей, эффективный адрес исключительной ситуации ключа и ключ памяти.

Сведения об аппаратном обеспечении

Действующий пользовательский набор ключей в выполняемом контексте поддерживающей ключи нити переводит действующий аппаратный регистр маски прав доступа (AMR) в формат, представленный абстрактным типом данных **ukeyset_t**. Эта информация предоставляется только для целей отладки. Для установки действующего пользовательского набора ключей следует применять только определенные программные службы.

- AMR - это 64-разрядный регистр, содержащий 32-разрядные пары (одна пара на ключ) для максимум 32 ключей, пронумерованных от 0 до 31.
  - Первый бит каждой пары представляет права на запись в соответствующий ключ.
  - Аналогично, второй бит каждой пары представляет права на чтение соответствующего ключа.
- Значение бита, равное 0, предоставляет соответствующий доступ, а значение, равное 1, запрещает доступ.
- Пара битов, предоставляющая права доступа для ключа 0, не управляется программой. Пользовательский ключ 0 является *пользовательским общим ключом*, и все нити всегда имеют полный доступ к данным в этом ключе, независимо от параметров действующего пользовательского набора ключей.
- Все остальные пары битов представляют *пользовательские частные ключи*, которые (если доступны) можно использовать для защиты ваших данных.

Пример программы

Ниже приведен пример программы,поддерживающей пользовательские ключи:

```
#include<stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys/ukeys.h>
#include <sys/syspest.h>
#include <sys/signal.h>
#include <sys/vminfo.h>

#define ROUND_UP(size,psize)    ((size)+(psize)-1 & ~((psize)-1))

/*
 * Это пример каркаса программы, поддерживающей пользовательские ключи.
 *
 * Структура private_data_1 отображает защищенную ключом область памяти,
 * к которой основная программа имеет свободный доступ, а "незащищенная"
 * команда имеет только права на чтение.
 */
struct private_data_1 {
        int some_data;
};
struct private_data_1 *p1;      /* указатель на структуру защищенных данных */

ukeyset_t keyset1RW;            /* набор ключей для предоставления доступа обработчику сигнала */

/*
 * Незащищенная функция должна успешно прочитать защищенные данные.
 *
 * Если значение счетчика равно 0, оно просто возвращается, так что инициатор может записать
 * увеличенное на единицу значение обратно в защищенное поле.
```

```
 *
 * Если значение счетчика равно 1, он пытается самостоятельно обновить защищенное поле.
 * Результатом должно являться SIGSEGV.
 */
int untrusted(struct private_data_1 *p1) {
        int count = p1->some_data;      /* Можно прочитать защищенные данные */
        if (count == 1)
                p1->some_data = count;  /* Однако запись запрещена */
        return count + 1;
}

/*
 * Обработчик сигнала обнаруживает умышленное нарушение защиты в
 * незащищенной функции выше при значении count == 1.
 * Обратите внимание, что обработчик вводится БЕЗ прав доступа к частным данным.
 */
void handler(int signo, siginfo_t *sip, void *ucp) {
        printf("siginfo: signo %d code %d\n", sip->si_signo, sip->si_code);
        (void)ukeyset_activate(keyset1RW, UKA_REPLACE_KEYS);
        exit(1);
}

main() {
        int nkeys;
        int pagesize = 4096;            /* аппаратный размер страницы данных */
        int padded_protsize_1;          /* размер страницы защищенных данных */
        struct vm_page_info page_info;

        ukey_t key1 = UKEY_PRIVATE1;
        ukeyset_t keyset1W, oldset;
        int rc;
        int count = 0;

        struct sigaction sa;

        /*
         * Попытка узнать о пользовательском ключе.
         */
        nkeys = ukey_enable();
        if (nkeys == -1) {
                perror("ukey_enable");
                exit(1);
        }
        assert(nkeys >= 2);

        /*
         * Определяется размер страницы области данных.
         */
        page_info.addr = (long)&p1;             /* адрес в области данных */
        rc = vmgetinfo(&page_info, VM_PAGE_INFO, sizeof(struct vm_page_info));

        if (rc)
                perror("vmgetinfo");
        else
                pagesize = page_info.pagesize;  /* получаем фактический размер страницы */

        /*
         * Необходимо выделить соответствующий странице объем памяти
         * для области, которую необходимо защитить ключом.
         */
        padded_protsize_1 = ROUND_UP(sizeof(struct private_data_1), pagesize);
        rc = posix_memalign((void **)&p1, pagesize, padded_protsize_1);
        if (rc) {
                perror("posix_memalign");
                exit(1);
        }
```

```
/*
 * Инициализация частных данных.
 * Это можно сделать до установки защиты.
 *
 * Обратите внимание, что указатель на частные данные находится в общей памяти.
 * Мы защищаем только сами данные.
 */
p1->some_data = count;

/*
 * Создание наборов ключей для применения для доступа к защищенной структуре.
 * Обратите внимание, что эти наборы ключей будут размещены в общей памяти.
 */
rc = ukeyset_init(&keyset1W, 0);
if (rc) {
        perror("ukeyset_init");
        exit(1);
}

rc = ukeyset_add_key(&keyset1W, key1, UK_WRITE);        /* WRITE */
if (rc) {
        perror("ukeyset_add_key 1W");
        exit(1);
}

keyset1RW = keyset1W;
rc = ukeyset_add_key(&keyset1RW, key1, UK_READ);        /* R/W */
if (rc) {
        perror("ukeyset_add_key 1R");
        exit(1);
}

/*
 * Доступ к частным данным запрещен с помощью применения частного ключа
 * к содержащей их странице (страницам).
 */
rc = ukey_protect(p1, padded_protsize_1, key1);
if (rc) {
        perror("ukey_protect");
        exit(1);
}

/*
 * Разрешить всему основному коду ссылаться на чтение/запись защищенных данных.
 */
oldset = ukeyset_activate(keyset1RW, UKA_ADD_KEYS);
if (oldset == UKSET_INVALID) {
        printf("ukeyset_activate failed\n");
        exit(1);
}

/*
 * Задается обработчик сигнала для SIGSEGV, для обнаружения преднамеренного
 * нарушения ключа в незащищенном коде.
 */
sa.sa_sigaction = handler;
SIGINITSET(sa.sa_mask);
sa.sa_flags = SA_SIGINFO;
rc = sigaction(SIGSEGV, &sa, 0);
if (rc) {
        perror("sigaction");
        exit(1);
}

/*
 * Основной цикл выполнения программы.
 */
```

```
        while (count < 2) {
                /*
                 * Когда необходимо выполнить "незащищенный" код, изменяем права доступа
                 * к частным данным на R/O, удалив права на запись.
                 */
                (void)ukeyset_activate(keyset1W, UKA_REMOVE_KEYS);

                /*
                 * Вызов незащищенной команды.  Она может только читать
                 * переданные ей защищенные данные.
                 */
                count = untrusted(p1);

                /*
                 * Восстанавливается полный доступ к частным данным.
                 */
                (void)ukeyset_activate(keyset1W, UKA_ADD_KEYS);

                p1->some_data = count;
        }
ukey_protect(p1, padded_protsize_1, UKEY_PUBLIC);
        free(p1);
        exit(0);
}
```

Поддержка больших программ

В этом разделе рассматриваются модели большого и сверхбольшого адресного пространства, применяемые для поддержки программ, данные которых нельзя разместить в рамках стандартной модели адресного пространства.

Модель большого адресного пространства поддерживается, начиная с AIX 4.3. Модель сверхбольшого адресного пространства поддерживается, начиная с AIX 5.1.

Примечание: Эта информация относится только к 32-разрядным процессам.

Виртуальное адресное пространство 32-разрядного процесса разделено на 16 областей (или *сегментов*) размером по 256 Мб, каждому из которых соответствует отдельный аппаратный регистр. Сегмент 2 (виртуальные адреса 0x20000000-0x2FFFFFFF) рассматривается операционной системой как *частный сегмент процесса*. По умолчанию в этом сегменте расположен пользовательский стек и данные, включая кучу. В частном сегменте процесса также находится защищенный блок, который применяется операционной системой и недоступен приложению.

Так как в одном сегменте хранятся пользовательские данные и данные стека, то их максимальный суммарный не превышает 256 Мб. Однако некоторые программы работают с большими областями данных (как инициализированными, так и нет) или захватывают большие фрагменты памяти с помощью функций **malloc** и **sbrk**. С помощью модели большого или сверхбольшого адресного пространства программы могут работать с данными, объем которых достигает 2 Гб.

Для применения модели большого или сверхбольшого адресного пространства в существующих программах необходимо указать ненулевое *значение maxdata*. Значение **maxdata** считывается из переменной среды **LDR_CNTRL** или из соответствующего поля в исполняемом файле. Некоторые программы жестко привязаны к стандартной модели адресного пространства и не будут работать с моделью большого адресного пространства.

Описание модели большого адресного пространства

Модель большого адресного пространства позволяет указанным программам работать с данными, объем которых превышает 256 Мб. Остальные программы будут продолжать работать со стандартной моделью. Для применения в программе модели большого адресного пространства укажите ненулевое значение

maxdata. Ненулевое значение **maxdata** можно задать с помощью команды **ld** при компиляции программы, либо экспортировав переменную среды **LDR_CNTRL** перед выполнением программы.

При запуске программы, применяющей модель большого адресного пространства, операционная система резервирует такое количество сегментов размером 256 Мб, какое необходимо для размещения данных, объем которых указан в параметре **maxdata**. Начиная с третьего сегмента, инициализированные данные программы считываются из исполняемого файла в память. Запись данных начинается с третьего сегмента даже в том случае, если значение **maxdata** меньше 256 Мб. В случае применения модели большого адресного пространства программа может использовать до 8 сегментов, либо до 2 или 3,25 Гб данных, соответственно.

В стандартной модели адресного пространства для функций **shmat** и **mmap** доступно 12 сегментов. В модели большого адресного пространства количество сегментов, доступных функциям **shmat** и **mmap**, уменьшается за счет сегментов, зарезервированных для данных. Так как максимальный объем данных составляет 2 Гб, функциям **shmat** и **mmap** всегда доступно не менее двух сегментов.

При применении модели большого адресного пространства пользовательский стек остается в сегменте 2. В этой связи максимальный размер стека составляет немногим менее 256 Мб. Однако приложение может переместить пользовательский стек в сегмент общей памяти или в выделенную память.

В то время как объем инициализированных данных в программе может быть достаточно большим, сохраняется ограничение на размер текста. В исполняемом файле, связанном с программой, суммарный размер раздела текста и раздела загрузчика не должен превышать 256 Мб. Такое ограничение связано с тем, что оба раздела размещаются в одном сегменте, предназначенном только для чтения (сегменте 1, или сегменте TEXT). Узнать размер разделов можно с помощью команды **dump**.

Описание модели сверхбольшого адресного пространства

Модель сверхбольшого адресного пространства во многом схожа с моделью большого адресного пространства, хотя существует и несколько отличий. Для применения в программе модели сверхбольшого адресного пространства укажите значение **maxdata** и свойство динамического выделения сегментов (dsa). Значение параметра **maxdata** и опции **DSA** можно задать с помощью команды **ld** или переменной среды **LDR_CNTRL**.

Если задано значение **maxdata**, то, как и в модели большого адресного пространства, данные программы записываются в память, начиная с третьего сегмента, и занимают необходимо число сегментов. Однако оставшиеся сегменты данных не резервируются для области данных в момент запуска, а выделяются динамически. Пока сегмент не требуется для области данных программы, он может применяться функциями **shmat** и **mmap**. В модели сверхбольшого адресного пространства программа может использовать не более 13 сегментов, или 3,25 Гб данных. 12 из этих сегментов, или 3 Гб памяти, доступно функциям **shmat** и **mmap**.

Если процесс пытается расширить свою область данных за счет добавления нового сегмента, то ему удается это сделать в том случае, если этот сегмент не применяется функциями **shmat** и **mmap**. Программа может освободить сегмент для области данных с помощью функций **shmdt** и **munmap**. После того как сегмент выделен для области данных, его назначение нельзя изменить, даже если размер области данных уменьшится.

Если опция dsa указана без значения **maxdata** (maxdata = 0), то применяется несколько иной алгоритм выделения памяти. Данные и стек процесса хранятся в сегменте 2, как в стандартной модели. Процессу недоступны глобальные общие библиотеки, поэтому все общие библиотеки, применяемые процессом, загружаются как частные. Преимущество такого алгоритма заключается в том, что все 13 сегментов (3,25 Гб) процесса могут применяться функциями **shmat** и **mmap**.

Для снижения вероятности применения в функциях **shmat** и **mmap** сегментов, которые могут быть заняты областью данных, в операционной системе действует другое правило выбора возвращаемого адреса (если не запрошен конкретный адрес). Обычно функции **shmat** и **mmap** возвращают адрес в нижнем из доступных сегментов. В модели сверхбольшого адресного пространства эти функции возвращают адрес в верхнем из

доступных сегментов. Если задан конкретный адрес, то он используется в том случае, если содержащий его сегмент еще не занят областью данных. Это справедливо для всех процессов, в которых задано свойство dsa.

В модели сверхбольшого адресного пространства для параметра **maxdata** допустимы значения от 0 до `0xD0000000`. Если указанное значение **maxdata** превышает `0xAFFFFFFF`, то программа не применяет глобальные общие библиотеки. Вместо этого все общие библиотеки загружаются как частные. В результате производительность программы может снизиться.

Применение моделей большого и сверхбольшого адресного пространства

Модель большого адресного пространства применяется в том случае, если задано ненулевое значение **maxdata** и не задано свойство динамического выделения сегментов (dsa). Модель сверхбольшого адресного пространства применяется при любом значении **maxdata**, если задана опция dsa. Значение **maxdata** и опцию dsa можно задать с помощью команды **ld** с флагом **-bmaxdata**.

Для того чтобы скомпоновать программу, в которой максимальный размер области данных будет составлять 8 сегментов, нужно ввести следующую команду:

```
cc -bmaxdata:0x80000000 sample.o
```

Для того, чтобы скомпоновать программу с применением модели сверхбольшого адресного пространства для системы Платформа с процессором POWER, введите следующую команду:

```
cc -bmaxdata:0xD0000000/dsa sample.o
```

Для того чтобы скомпоновать программу с применением модели сверхбольшого адресного пространства, введите следующую команду:

```
cc -bmaxdata:0xD0000000/dsa sample.o
```

Для применения модели большого или сверхбольшого адресного пространства в существующих программах можно указать значение **maxdata** с помощью переменной среды **LDR_CNTRL**. Например, для запуска программы **a.out**, для области данных которой будет зарезервировано 8 сегментов, нужно ввести следующую команду:

```
LDR_CNTRL=MAXDATA=0x80000000 a.out
```

Следующая команда запускает программу **a.out** с использованием модели сверхбольшого адресного пространства, в которой данные программы могут занимать до 8 сегментов памяти:

```
LDR_CNTRL=MAXDATA=0x80000000@DSA a.out
```

Существующую программу также можно настроить на применение модели большого или сверхбольшого адресного пространства. Для того чтобы изменить значение **maxdata** в существующей 32-разрядной программе XCOFF **a.out** на `0x80000000`, введите следующую команду:

```
/usr/ccs/bin/ldedit -bmaxdata:0x80000000 a.out
```

Если в существующей 32-разрядной программе XCOFF **a.out** со значением **maxdata**, равным 0x80000000, еще не задано свойство **DSA**, его можно добавить с помощью следующей команды:

```
/usr/ccs/bin/ldedit -bmaxdata:0x80000000/dsa a.out
```

Узнать значение **maxdata** и определить наличие параметра dsa можно с помощью команды **dump**.

Некоторые программы жестко привязаны к стандартной модели адресного пространства. Если в переменной среды **LDR_CNTRL** или в исполняемом файле было задано ненулевое значение **maxdata**, эти программы не будут работать.

Выполнение программ с большими областями данных

При запуске программы, применяющей модель большого адресного пространства, операционная система пытается изменить гибкое ограничение на размер данных таким образом, чтобы оно совпадало со значением **maxdata**. Если значение **maxdata** *превышает* текущее жесткое ограничение на размер области данных, то программа не будет запущена (если переменной среды **XPG_SUS_ENV** присвоено значение ON) либо гибкое ограничение будет установлено на уровне текущего жесткого ограничения.

Если значение **maxdata** *меньше* объема статических данных программы, то программа не будет запущена.

После размещения инициализированных и неинициализированных данных программы в сегментах, начиная с третьего, система вычисляет положение разделителя. Разделитель отделяет статические данные процесса от данных, память для которых выделяется динамически. С помощью функций **malloc**, **brk** и **sbrk** процесс может увеличивать область данных, перемещая разделитель.

Например, если в программе задано значение **maxdata**, равное `0x68000000`, то максимальное значение разделителя находится в середине сегмента 9 (`0x98000000`). Функция **brk** позволяет перенести разделитель за границу сегмента при условии, что размер области данных не будет превышать текущее гибкое ограничение.

Функция **setrlimit** позволяет процессу установить любое гибкое ограничение на размер области данных, не превышающее жесткое ограничение. Однако максимальный размер области данных ограничен исходным значением **maxdata**, округленным до числа, кратного 256 Мб.

Большинство функций не зависят от объема данных программы. Большой объем данных больше всего сказывается на работе функций **shmat** и **mmap**, так как у них остается меньше сегментов. Дочерние процессы программы, применяющей модель большого адресного пространства, наследуют текущие ограничения на объем данных.

Особые рекомендации

Программа с большим объемом данных требует большого объема пространства подкачки. Например, если программа с адресным пространством размером 2 Гб обращается к каждой странице этого адресного пространства, то объем пространства подкачки в системе должен составлять 2 Гб. Если пространство подкачки будет переполнено, операционная система завершит некоторые процессы. Поскольку программы с большим объемом данных обычно используют большой объем пространства подкачки, они завершаются в первую очередь.

Отладка программ с большим объемом данных не отличается от отладки других программ. Команда **dbx** может выполнять отладку большой программы, когда она активна, либо на основании дампа памяти. Дамп памяти программы с большим объемом данных может достигать очень больших размеров. Для того чтобы файлы дампа не усекались, должно быть задано достаточно большое ограничение **coredump**, а в файловой системе, применяемой программой, должно быть достаточно свободного пространства.

В некоторых приложениях предполагается, что применяется стандартная модель адресного пространства. В случае применения модели большого или сверхбольшого адресного пространства такие программы могут не работать. Не задавайте переменную **LDR_CNTRL** при применении этих программ.

Если применяется модель сверхбольшого адресного пространства, то для перемещения разделителя пространства адресации с шагом, превышающим 2 Гб, требуется изменить код программы. Это связано с тем, что в параметре системного вызова **sbrk** указывается значение со знаком. Для перемещения разделителя в необходимое положение функцию **sbrk** можно вызвать несколько раз.

Информация, связанная с данной:

brk

exec

fork

malloc

setrlimit

Команда dd

Команда ld

Формат объектного файла XCOFF (a.out)

Программирование в многопроцессорных системах

В однопроцессорной системе нити обрабатываются поочередно: каждой из них выделяется квант времени. В многопроцессорных системах несколько нитей обрабатываются одновременно - каждый процессор обрабатывает отдельную нить. Обработка нескольких нитей процесса на разных процессорах иногда позволяет повысить общую производительность. Однако преимущества многопроцессорных систем проявляются только в случае программ с несколькими нитями.

Большинство пользователей может не задумываться о том, сколько процессоров в системе, так как за организацию работы в многопроцессорной среде отвечает операционная система и ее программы. Пользователи могут связать свои процессы с процессорами (тогда эти процессы будут обрабатываться только на указанных процессорах), однако это делать не обязательно и, как правило, не нужно. Даже для большинства разработчиков преимущества многопроцессорных систем сводятся к возможности разбиения процесса на несколько нитей. С другой стороны, разработчики компонентов ядра вынуждены учитывать некоторые особенности многопроцессорных систем при создании программ и переносе готовых программ в такие системы.

Идентификация процессоров

Симметричные многопроцессорные системы (SMP) поддерживают несколько плат CPU, на каждой из которых может быть установлено по два процессора.

Например, в системе с четырьмя процессорами устанавливается две платы CPU с двумя процессорами на каждой. В командах, функциях и сообщениях, относящихся к процессорам, необходимо так или иначе идентифицировать процессор. Процессоры можно различать по физическим и логическим номерам, а также по их именам в ODM и кодам расположения.

Имена процессоров ODM

ODM - это система, применяемая для идентификации различных компонентов компьютера, в том числе адаптеров и периферийных устройств, таких как принтеры, терминалы, диски, карты памяти и процессоров.

ODM создает имена процессоров и карт процессоров путем добавления номера к префиксу `cpucard` или `proc` (нумерация начинается с 0). Таким образом, первой карте процессоров будет присвоено имя `cpucard0`, а второму процессору на этой карте - имя `proc1`.

Коды расположения ODM для процессоров состоят из четырех двухсимвольных полей и задаются в виде *AA-BB-CC-DD*, где:

Код	Описание
AA	Всегда равно `00`. Это значение соответствует системному блоку.
BB	Задает номер карты процессоров. Возможные значения: `0P`, `0Q`, `0R` или `0S`, что соответствует первой, второй, третьей и четвертой карте процессоров.
CC	Всегда равно `00`.
DD	Указывает положение процессора на карте процессоров. Возможные значения: `00` или `01`.

Логические номера процессоров

Процессорам также присваиваются логические номера, последовательность которых начинается с 0 (нуля). Логические номера присваивается только включенным процессорам.

Логический номер процессора 0 (ноль) соответствует первому включенному физическому процессору, логический номер 1 - второму и т.д. Обычно во всех командах операционной системы и библиотечных функциях процессоры идентифицируются по логическим номерам.

Состояния процессоров ODM

Правильно работающие процессоры можно включать и отключать программно, с помощью команд. Если в процессоре обнаружена аппаратная неполадка, то он помечается как **неисправный**. ODM различает следующие состояния процессоров:

Состояние	Описание
разрешено	Процессор исправен и может использоваться системой AIX.
отключена	Процессор исправен, но недоступен системе AIX.
faulty	Процессор неисправен (обнаружена аппаратная неполадка).

Управление работой процессоров

В этом разделе приведены инструкции по управлению использованием процессоров в системе с несколькими процессорами.

В многопроцессорной системе предусмотрены следующие способы управления загрузкой процессоров:

- Пользователь может связать процесс или нить ядра с конкретным процессором.

Привязка процессов и нитей ядра к процессору

Пользователи могут указать, что их процессы должны выполняться на определенном процессоре. Такое действие называется *привязкой*. Системный администратор может связать любой процесс с любым процессором. Для этого предусмотрена команда **bindprocessor**.

С процессором связан не сам процесс, а его нити ядра. Это означает, что нити ядра всегда будут выполняться на одном и том же процессоре до тех пор, пока связь не будет удалена. При создании новой нити она связывается с тем же процессором, что и ее родительская нить.

Это относится к главной нити нового процесса, создаваемого с помощью вызова **fork**. Новая нить будет связана с тем же процессором, что и нить, вызвавшая **fork**. При вызове функции **exec** привязка к процессору не наследуется. Если процесс связан с процессором, то все его дочерние процессы будут связаны с тем же процессором, если это не будет явно переопределено.

Для создания связи процесса с процессором нужно указать логический номер целевого процессора. Таким образом, процесс может быть связан только с включенным процессором. Логические номера процессоров можно просмотреть с помощью команды **bindprocessor -q**. В системе с четырьмя включенными процессорами эта команда выдаст примерно следующую информацию:

```
Доступные процессоры: 0 1 2 3
```

Из программы нить можно связать с процессором с помощью процедуры **bindprocessor**. Она позволяет связать с процессором выбранную или все нити ядра. Кроме того, из программы можно удалить связь нити с процессором.

Работа с динамическое отключение процессоров

Начиная с систем 7044 модели 270, аппаратное обеспечение всех компьютеров с более чем двумя процессорами может автоматически регистрировать исправимые ошибки с помощью микрокода. Если число таких ошибок невелико, то они не представляют серьезной опасности и их можно игнорировать. Однако если число ошибок одного из процессоров продолжает увеличиваться, это может говорить о том, что в ближайшем будущем возникнет неустранимый сбой данного компонента системы. Подобный прогноз составляется микрокодом на основании числа ошибок и пороговых значений.

AIX осуществляет непрерывный контроль за работой аппаратного обеспечения и отслеживает аппаратные ошибки, регулярно опрашивая встроенное программное обеспечение. Когда число ошибок процессора становится больше порогового значения, и встроенное программное обеспечения делает прогноз о его скором выходе из строя, оно отправляет отчет об ошибках операционной системе AIX и заносит информацию об ошибке в системный протокол ошибок. В многопроцессорных системах при возникновении ошибок определенного типа AIX может отказаться от использования ненадежного процессора. Эта функция называется *динамическим отключением процессора*.

Встроенное программное обеспечение помечает процессор для его отключения при последующих перезагрузках системы. Он будет находиться в таком состоянии до тех пор, пока его не заменят.

Возможное влияние на приложения

Отключение процессора никак не влияет на большую часть приложений, включая драйверы и расширения ядра. Тем не менее, с помощью стандартных интерфейсов AIX можно определить, работает ли приложение или расширение ядра в многопроцессорной системе, узнать число процессоров и связать нити с определенными процессорами.

Интерфейс bindprocessor связывания процессов и нитей с процессорами использует номера связывания CPU. Номера связывания CPU находятся в диапазоне от 0 до *N*-1, где *N* - общее число процессоров. Поскольку приложения и расширения ядра не рассчитаны на наличие пропусков в нумерации процессоров, AIX всегда имитирует отключение последнего процессора (с наибольшим номером). Например, в 8-процессорной системе SMP применяются номера CPU [0..7]. В случае сбоя одного из процессоров в системе останется 7 доступных CPU с номерами от 0 до 6. С внешней точки зрения это выглядит так, как будто был удален седьмой процессор, вне зависимости от того, какой физический процессор был отключен.

Примечание: В дальнейшем словом *CPU* будет обозначаться логический ресурс, а словом *процессор* - физический ресурс.

Если из-за отключения процессора AIX без предупреждения аннулирует связь нитей с процессором или переместит нити на другой процессор, в приложениях и расширениях ядра могут возникнуть ошибки. Функция динамического отключения процессора предоставляет программные интерфейсы, с помощью которых приложения и расширения ядра могут получить уведомление о планируемом отключении процессора. Получив такое уведомление, приложения и расширения ядра должны перенести свои нити и ресурсы (например, таймеры) с того CPU, которому назначен наибольший номер, и адаптироваться к новой конфигурации.

Если, несмотря на уведомление, некоторые приложения и расширения ядра не перенесут свои нити с последнего CPU, процессор не будет отключен. В этом случае AIX занесет в протокол сообщение об аварийном завершении отключения и продолжит использовать неисправный процессор. Сбой процессора вызовет сбой всей системы. В связи с этим все приложения и расширения ядра, связывающие свои нити с CPU, обязательно должны получать и обрабатывать сообщения о предстоящем отключении процессора.

В тех редких случаях, когда системе не удается отключить процессор, она отправляет системным администраторам дополнительное сообщение. Ознакомившись с записью, занесенной в протокол ошибок, системный администратор может запланировать обслуживание системы и заменить ненадежный компонент до того, как система полностью выйдет из строя.

Процедура динамического отключения процессора

Обычно процедура отключения процессора выглядит следующим образом:

1. Встроенное программное обеспечение обнаруживает, что один из процессоров превысил допустимое число устранимых ошибок.
2. AIX заносит сообщение об ошибках процессора в протокол ошибок и начинает процесс отключения процессора.

3. AIX отправляет уведомление пользовательским процессам и нитям, связанным с CPU с максимальным логическим номером.
4. AIX ждет, пока с последнего CPU не будут перенесены все связанные нити. Если через десять минут с процессором все еще будут связаны какие-либо нити, AIX отменяет динамическое отключение процессора. В противном случае, AIX вызывает зарегистрированные ранее обработчики HAEH. Если HAEH возвратит сообщение об ошибке, то отключение будет отменено. В противном случае AIX продолжит процесс отключения и остановит неисправный процессор.

Если во время отключения возникнет сбой, AIX занесет в протокол информацию об ошибке, указав причину, по которой отключение не было выполнено. Системный администратор может просмотреть протокол ошибок, выполнить необходимые действия по устранению ошибки и повторить попытку отключения. Например, если отключение процесса было прервано из-за того, что одно из приложений не аннулировало связь своих нитей с процессором, системный администратор должен остановить приложение, снова запустить процедуру отключения процессора, а после ее успешного завершения - перезапустить приложение.

Программные интерфейсы для работы с отдельными процессорами

В следующих разделах описаны доступные программные интерфейсы:

Интерфейсы для определения количества CPU в системе

sysconf, функция

Функция **sysconf** возвращает количество процессоров в следующих параметрах:

- **_SC_NPROCESSORS_CONF**: Количество настроенных процессоров
- **_SC_NPROCESSORS_ONLN**: Количество включенных процессоров

Дополнительная информация приведена в описании функции **sysconf** в *Technical Reference: Base Operating System and Extensions, Volume 2*.

Значение, возвращаемое функцией **sysconf** в параметре `_SC_NPROCESSORS_CONF`, не изменяется после перезагрузки системы. В системах с одним процессором (UP) это значение равно 1. Значения, превышающие 1, относятся к многопроцессорным системам (MP). Значение, возвращаемое в параметре **_SC_NPROCESSORS_ONLN**, задает количество активных CPU. Оно уменьшается на единицу при каждом отключении процессора.

Поле **_system_configuration.ncpus** задает число активных CPU в системе. Это поле аналогично параметру **_SC_NPROCESSOR_ONLN**. Дополнительная информация приведена в разделе Файл **systemcfg.h** книги *Справочник по файлам* .

Для того чтобы в приложении можно было узнать, сколько процессоров было доступно в момент загрузки, в таблицу `_system_configuration` было добавлено поле **ncpus_cfg**. Значение этого поля не изменяется между перезагрузками системы.

CPU идентифицируются по номерам CPU для связывания, которые образуют диапазон [0..(ncpus-1)], где ncpu - общее количество CPU. У процессора также есть номер физического CPU, зависящий от расположения процессора на карте адаптера, карты адаптера и т. п. Команды и функции, работающие с номерами CPU, всегда используют номера CPU для связывания. Количество CPU может изменяться во время работы системы, однако их номера для связывания всегда образуют непрерывный диапазон значений [0..(ncpus-1). В результате с точки зрения пользователя при динамическом отключении процессора всегда удаляется CPU с максимальным номером для связывания ("последний" CPU), независимо от того, на каком физическом процессоре на самом деле произошел сбой.

Примечание: Наибольший номер CPU для связывания можно узнать с помощью переменной *ncpus_cfg*.

Интерфейсы связывания нитей с процессорами

Команда **bindprocessor** и программные интерфейсы **bindprocessor** позволяют связать нить или процесс с конкретным CPU, указав номер для связывания этого CPU. Оба интерфейса позволяют связывать нити и процессы только с активными CPU. Программы, напрямую вызывающие интерфейс **bindprocessor**, а также программы, связанные с процессором с помощью команды **bindprocessor**, должны выполнять необходимые действия при отключении процессора.

Основной ошибкой, встречающейся при отключении процессора, является попытка связать нить или процесс с отключенным процессором. Программы, вызывающие **bindprocessor**, всегда должны проверять значение, возвращенное этой функцией.

Дополнительная информация об этих интерфейсах приведена в разделе Команда **bindprocessor** книги *Справочник по командам, том 1* и Функция **bindprocessor** книги *Technical Reference: Base Operating System and Extensions, Volume 1*.

Интерфейсы уведомления об отключении процессора

Для пользовательских приложений, нити которых связаны с последним процессором, и расширений ядра применяется разный механизм оповещения.

Оповещение пользовательских программ

Всем нитям пользовательского приложения, связанным с последним CPU, отправляются сигналы **SIGCPUFAIL** и **SIGRECONFIG**. Приложения нужно изменить таким образом, чтобы они получали эти сигналы и удаляли с последнего CPU свои нити (путем аннулирования связи нитей с этим CPU или путем переноса нитей на другой CPU).

Оповещение расширений ядра

Драйверы и расширения ядра, которые должны уведомляться о планируемом отключении процессора, должны зарегистрировать в ядре функцию обработчика HAEH. Эта функция будет вызвана при отключении процессора. При удалении расширения ядра можно отменить регистрацию этой функции HAEH с помощью специального интерфейса.

Регистрация обработчика haeh

Для уведомления расширений ядра о событиях, влияющих на уровень готовности системы, ядро экспортирует новую функцию.

Формат ее вызова следующий:

```
int register_HA_handler(ha_handler_ext_t *)
```

Более подробная информация о вызове этой функции приведена в разделе **register_HA_handler** книги *Управление операционной системой и устройствами*.

При успешном завершении эта функция возвращает значение 0. Ненулевое значение свидетельствует о сбое.

В качестве аргумента функции передается указатель на структуру, описывающую HAEH расширения ядра. Эта структура определена в заголовочном файле **sys/high_avail.h** следующим образом:

```
typedef struct
_ha_handler_ext_ {
    int (*_fun)();        /* Вызываемая функция */
    long long _data;      /* Частные данные для (*_fun)() */
    char        _name[sizeof(long long) + 1];
} ha_handler_ext_t;
```

Расширение ядра может при необходимости использовать частное поле **_data**. Значение, заданное в этом поле при регистрации, будет передано зарегистрированной функции в качестве параметра при обработке отключения процессора.

Поле **_name** содержит строку длиной не более 8 символов (не включая последний символ NULL), которая однозначно идентифицирует расширение ядра. Это имя не должно совпадать с именами других зарегистрированных расширений ядра. Если при вызове функции HAEH расширения ядра произойдет ошибка, имя расширения ядра будет записано в поле подробной информации записи протокола ошибок CPU_DEALLOC_ABORTED.

Расширения ядра могут зарегистрировать свою функцию HAEH только один раз.

Вызов обработчика haeh

Параметры, вызывающие функцию HAEH:

- Значение поля **_data** структуры **ha_handler_ext_t**, переданной функции **register_HA_handler**.
- Указатель на структуру **ha_event_t**, определенную в файле **sys/high_avail.h** следующим образом:

```
typedef struct {                    /* Событие HAEH */
    uint _magic;                    /* Тип события */
#define HA_CPU_FAIL 0x40505546      /* "CPUF" */
    union {
        struct {                   /* Сбой процессора */
            cpu_t dealloc_cpu;     /* номер неисправного CPU для связывания */
                       ushort domain;         /* будущее расширение */
            ushort nodeid;         /* будущее расширение */
            ushort reserved3;      /* будущее расширение */
            uint reserved[4];      /* будущее расширение */
        } _cpu;
        /* ... */                  /* Дополнительные типы событий -- */
        /* будущее расширение */
    } _u;
} haeh_event_t;
```

Функция должна возвращать один из следующих кодов, определенных в файле **sys/high_avail.h**:

```
#define HA_ACCEPTED 0     /* Успешное выполнение */
#define HA_REFUSED -1     /* Ошибка */
```

Если хотя бы одно из зарегистрированных расширений не возвратит HA_ACCEPTED, отключение процессора будет прервано. Функции HAEH вызываются в среде процесса и не требуют закрепления в памяти.

Если расширение ядра зависит от конфигурации CPU, связанная с ним процедура HAEH должна правильно обрабатывать предстоящее отключение процессора. Конкретная процедура обработки зависит от характера приложения. Однако для отключения процессора операционной системе AIX достаточно, чтобы приложение переместило свои нити, связанные с последним CPU. Кроме того, если приложение использовало таймеры, запущенные из связанных нитей, такие таймеры также следует связать с другим CPU. Если работа программы зависит от того, с каким именно процессором связан таймер, нужно выполнить необходимое действие над таймером (например, остановить) и перезапустить его после связывания нитей с другим CPU.

Отмена регистрации обработчика haeh

Для обеспечения целостности системы и предотвращения ее сбоя завершающие свою работу расширения ядра должны отменять регистрацию HAEH. Для этого служит следующий интерфейс:

```
int unregister_HA_handler(ha_handler_ext_t *)
```

В случае успешного завершения этот интерфейс возвращает значение 0. Ненулевое значение означает, что произошла ошибка.

Более подробная информация о вызове этой функции приведена в разделе **unregister_HA_handler** в *Technical Reference: Kernel and Subsystems, Volume 1* .

Отключение процессора в тестовой среде

Для тестирования приложений и расширений ядра, которые должны обрабатывать удаление процессоров, предусмотрена команда, вызывающая отключение CPU с указанным логическим номером. Формат ее вызова следующий:

```
cpu_deallocate номер-CPU
```

где:

номер-CPU - это логический номер CPU.

Для включения отключенного таким образом процессора необходимо перезагрузить систему. Эта команда предназначена только для тестирования и *не* должна использоваться системными администраторами в других целях.

Динамическая защита памяти

Одной из конструктивных особенностей систем AIX является устойчивость к ошибкам памяти. Устойчивость к ошибкам памяти обеспечивается путем восстановления на уровне аппаратного обеспечения и операционной системы.

Ошибки памяти можно классифицировать различными способами, однако в рамках данного обсуждения они разделяются на исправимые и неисправимые.

Исправимые ошибки позволяют обратиться к данным в конкретных расположениях; неисправимые ошибки - приводят к потере данных. Неисправимые ошибки, как правило, устраняются путем аппаратного обеспечения избыточности данных в подсистеме памяти, либо путем маскировки поврежденной области памяти в ходе загрузки операционной системы.

Устойчивость AIX к ошибкам памяти представляет собой средство предотвращения перехода ошибок памяти из категории исправимых в категорию неисправимых ошибок памяти с помощью метода динамической защиты памяти. Динамическая защита памяти обеспечивается функциями, предоставляемыми аппаратным обеспечением. Аппаратное обеспечение предоставляет способы обнаружения и исправления ошибок (такие как очистка памяти и контуры коррекции ошибок (ECC)). Кроме того, аппаратное обеспечение может поддерживать функции предотвращения возникновения неисправимых ошибок, в частности избыточное управление битами.

В дополнение к рассмотренным механизмам аппаратное обеспечение может предоставлять операционной системе информацию об ошибках, наиболее эффективно обрабатываемых с помощью метода динамической защиты памяти. Для этой цели указываются области памяти для освобождения. Операционная система AIX использует эту информацию для маскировки областей памяти с запретом их применения. Операционная система перемещает данные, содержащиеся в поврежденной области памяти, в другую область памяти и запрещает использование страницы памяти, содержащей поврежденную область памяти. Операции защиты памяти выполняются операционной системой без вмешательства пользователя незаметно для конечных пользователей и приложений.

Службы блокировки с поддержкой многопроцессорных систем

Программисту может понадобиться создать собственные высокоуровневые механизмы блокировки, которые будут использоваться вместо стандартных служб взаимной блокировки из библиотек нитей.

Например, если система работы с базами данных уже использует некоторый набор внутренних служб блокировки, может оказаться проще изменить эти службы для работы в новой системе, чем переделывать все внутренние модули для работы со стандартными службами.

В этих целях AIX предоставляет набор атомарных служб блокировки, на основе которых могут быть созданы службы блокировки более высокого уровня. Для создания служб блокировки с поддержкой многопроцессорных систем (таких как стандартные взаимные блокировки) следует применять атомарные службы блокировки, описанные в этом разделе, а не атомарные операции, такие как **compare_and_swap**.

Службы блокировки с поддержкой многопроцессорных систем

Службы блокировки предназначены для организации поочередного доступа к ресурсам, которые могут использоваться одновременно. Например, службы блокировки могут применяться при вставке элементов в список, во время которой выполняется обновление нескольких указателей. Если обновление указателей одним процессом будет прервано аналогичной операцией в другом процессе, может возникнуть ошибка. Последовательность операций, которая не должна прерываться подобным образом, называется *критическим участком*.

Службы блокировки применяют слово блокировки для обозначения состояния блокировки: 0 (ноль) обозначает свободный ресурс, а 1 (один) - занятый. Таким образом, при выполнении блокировки объекта служба должны выполнить следующие действия:

```
определить значение слова блокировки
если ресурс свободен
        установить состояние "занято"
        вернуть состояние успешного выполнения (SUCCESS)
...
```

Так как такая последовательность операций (чтение, сравнение, установка) сама по себе создает критический участок, необходимо выполнять их в особом режиме. В однопроцессорной системе непрерывное выполнение этих операций гарантируется механизмом запрета прерываний. Однако в многопроцессорной системе атомарная операция проверки и установки должна обеспечиваться аппаратно - обычно ей соответствует машинная инструкция. Кроме того, для временной блокировки других операций чтения и записи применяются особые инструкции синхронизации, называемые *ограничениями на импорт и экспорт* - их реализация зависит от процессора. Такие инструкции предотвращают одновременный доступ нескольких процессоров, а также запрещают изменение порядка операций чтения-записи, применяемое в современных процессорах. Они определены следующим образом:

Ограничения на импорт
: *Ограничение на импорт* - это специальная машинная инструкция, обеспечивающая задержку до тех пор, пока не будут выполнены все ранее вызванные инструкции. Ее применение совместно с блокировкой предотвращает фиктивное выполнение инструкций до тех пор, пока не будет получена блокировка.

Ограничения на экспорт
: *Ограничение на экспорт* гарантирует, что защищаемые данные видны всем остальным процессорам до освобождения блокировки.

Для того чтобы скрыть подобные сложности и обеспечить независимость этих механизмов от реализации в конкретной системе, предусмотрены следующие процедуры:

_check_lock
: Обеспечивает атомарную проверку и изменение переменной размером в одно слово с использованием *ограничения на импорт* в многопроцессорных системах. Функция **compare_and_swap** выполняет аналогичные действия, но не использует ограничение на импорт, и поэтому не может применяться для реализации блокировки.

_clear_lock
: Обеспечивает атомарное изменение переменной размером в одно слово с использованием *ограничения на экспорт* в многопроцессорных системах.

Программирование процессов ядра

Полная информация о программировании процессов ядра приведена в разделе *Kernel Extensions and Device Support Programming Concepts*. В этом разделе обсуждаются основные особенности, которые следует учитывать при программировании процессов ядра в многопроцессорных системах.

При обращении к ряду важных ресурсов часто необходима сериализация. Для сериализации обращений нитей к ресурсам в среде процесса можно применять функции блокировки, однако они не обеспечивают защиту от одновременного доступа в среде прерываний. Во всех новых и перенесенных программах вместо службы ядра **i_disable** должны применяться службы **disable_lock** и **unlock_enable**, которые помимо управления прерываниями применяют простые блокировки. Эти службы можно применять и в однопроцессорных системах: в этом случае они используют только службы прерываний, без блокировок. Дополнительная информация приведена в разделе Службы блокировки ядра in *Kernel Extensions and Device Support Programming Concepts*.

Драйверы устройств по умолчанию запускаются в логической однопроцессорной среде, в так называемом *последовательном* режиме. Большинство хороших драйверов для однопроцессорных систем работают в этом режиме без какой-либо модификации, однако для наиболее полного использования преимуществ многопроцессорной среды их необходимо тщательно проверить и при необходимости изменить. Наконец, службы ядра, предназначенные для работы с таймерами, теперь должны возвращать значения, поскольку в многопроцессорной среде работа этих служб не всегда завершается успешно. Следовательно, в новом или перенесенном коде необходимо предусмотреть проверку возвращаемых значений этих функций. Дополнительная информация приведена в разделе Using Multiprocessor-Safe Timer Services в *Kernel Extensions and Device Support Programming Concepts*.

Примеры служб блокировки

Процедуры блокировки с поддержкой многопроцессорных систем могут применяться для создания высокоуровневых процедур блокировки без использования библиотеки нитей. В этом примере показана частичная реализация процедур, аналогичных процедурам **pthread_mutex_lock** и **pthread_mutex_unlock** из библиотеки нитей.

```
#include <sys/atomic_op.h>  /* элементарные операции блокировки */
#define SUCCESS          0
#define FAILURE          -1
#define LOCK_FREE        0
#define LOCK_TAKEN       1

typdef struct {
        atomic_p         lock;   /* слово блокировки */
        tid_t            owner;  /* идентификатор владельца блокировки */
        ...              /* поля, относящиеся к конкретной реализации */
} my_mutex_t;

...

int my_mutex_lock(my_mutex_t *mutex)
{
tid_t   self;   /* идентификатор вызывающей нити */

        /*
        Выполнение различных проверок:
          допустимо ли значение указателя взаимной блокировки?
          инициализирована ли взаимная блокировка?
        */
        ...

        /* проверка того, что взаимная блокировка не принадлежит
                                            вызывающей нити */
        self = thread_self();
        if (mutex->owner == self)
                return FAILURE;

        /*
```

```
        Выполнение атомарной операции проверки и установки в цикле.
        В этой версии - освобождать процессор при неудачной попытке.
        В другом варианте возможна непрерывная проверка
                 или освобождение процессора после фиксированного числа попыток.
        */
        while (_check_lock(mutex->lock, LOCK_FREE, LOCK_TAKEN))
                yield();

        mutex->owner = self;
        return SUCCESS;
} /* конец my_mutex_lock */

int my_mutex_unlock(my_mutex_t *mutex)
{
        /*
        Выполнение различных проверок:
          допустимо ли значение указателя взаимной блокировки?
          инициализирована ли взаимная блокировка?
        */
        ...

        /* проверка того, что взаимная блокировка не принадлежит
                                            вызывающей нити */
        if (mutex->owner != thread_self())
                return FAILURE;

        _clear_lock(mutex->lock, LOCK_FREE);
        return SUCCESS;
} /* конец my_mutex_unlock */
```

Информация, связанная с данной:

Блокировка служб ядра

Применение служб таймера, безопасных для микропроцессора

bindprocessor

compare_and_swap

pthread_mutex_unlock

disable_lock

i_disable

unlock_enable

bindprocessor, команда

Функция динамической трассировки ProbeVue

Средство динамической трассировки ProbeVue можно применять как для анализа производительности, так и для отладки. В ProbeVue используется язык программирования Vue для динамического описания точек трассировки и действий, выполняемых в этих точках.

ProbeVue обладает следующими функциями:

- Отсутствие предварительно скомпилированных точек трассировки. ProbeVue не требует изменения ядра и пользовательских приложений.
- Перехватчики трассировки необязательно заранее компилировать. Они компилируются как часть исходной программы. ProbeVue не требует изменения ядра и пользовательских приложений.
- Пока перехватчики трассировки динамически не включены, они не работают (не существуют).
- Операции трассировки (указанные инструментальным кодом), выполняемые в перехватчике трассировки, подгружаются динамически во время включения перехватчика.
- Данные трассировки, полученные в рамках действий трассировки, доступны для немедленного просмотра и могут отображаться в виде вывода на терминал или сохраняются в файл для последующего просмотра.

Примечание: dbx и Probevue не могут отлаживать процесс одновременно. Иногда отладка процесса, запущенного ProbeVue, может привести к блокировке dbx во время его подключения к процессу.

Концепции ProbeVue

В процессе трассировки или тестирования в ProbeVue текущие глобальные и контекстные значения записываются в буфер трассировки.

Собранная информация называется данными трассировки. Система предоставляет средства для чтения данных из буфера трассировки и передачи их пользователям системы.

Точка тестирования обозначает точку системной операции, где можно выполнять трассировку. При выполнении динамической трассировки, тесты подключаются только к тем точкам тестирования, которые трассируются. *Включение* теста — это операция подключения теста к точке тестирования, *выключение* теста — операция отключения теста от точки тестирования. Тест *срабатывает* или *активируется*, когда выполнение системной операции доходит до включенной точки тестирования, это приводит к выполнению действий трассировки.

ProbeVue поддерживает две больших категории точек тестирования.

Расположение теста
: Расположение кода режима пользователя или режима ядра, где выполняется действие трассировки. Включенные тесты в расположении теста срабатывают, когда нить, выполняющая код, доходит до их расположения.

Событие теста
: Интересующее событие, при наступлении которого выполняется действие трассировки. События теста не просто отображаются на определенное место кода. Включенные тесты, обозначающие событие теста, настраиваются на срабатывание при наступлении этого события.

Кроме этого, ProbeVue различает точки тестирования по типу. Тип теста обозначает набор точек тестирования, имеющих общие параметры, например, точки тестирования, обозначающие вход и выход системных вызовов, или точки тестирования, обозначающие изменение системной статистики. Различение тестов по типам создает основу для широкого спектра точек тестирования.

Команда ProbeVue

Команда probevue запускает сеанс динамической трассировки или сеанс ProbeVue.

Команда **probevue** начинает сеанс динамической трассировки или сеанс ProbeVue. Команда **probevue** получает на входе сценарий Vue из файла или командной строки и активирует сеанс ProbeVue. В зависимости от параметров командной строки данные трассировки, собранные во время сеанса ProbeVue, могут быть выведены на терминал или сохранены в указанном файле.

Сеанс ProbeVue остается активным до нажатия сочетания клавиш Ctrl-C в терминале или выполнения операции exit сценарием Vue.

Каждый вызов **probevue** активирует отдельный сеанс динамической трассировки. Одновременно могут быть активны несколько сеансов, но каждый сеанс представляет только те данные данные трассировки, которые собраны в нем. Параллельные сеансы обычно ничего не знают друг о друге.

Выполнение команды **probevue** является привилегированной операцией, поэтому обычным пользователям требуются дополнительные права для начала сеанса трассировки.

Язык программирования Vue

Язык программирования Vue используется для создания собственных спецификаций тестирования для ProbeVue.

Язык программирования Vue используется для создания собственных спецификаций тестирования для ProbeVue. Сценарий Vue или программа Vue — программа, написанная на языке Vue. Сценарии Vue используются для следующих целей:

- Идентификация точек тестирования, где должен динамически включаться тест.
- Описание условий активации теста.
- Описание выполняемых действий, включая данные трассировки, которые требуется собрать.

Если коротко; то сценарий Vue сообщает ProbeVue, где, когда и что трассировать. Сценарии Vue имеют расширение ".e", отличающее их от файлов других типов.

Примечание: Vue является одновременно и языком программирования и языком сценариев. Это специализированный язык динамической трассировки. Vue поддерживает подмножество языка C и сценарный синтаксис, наиболее подходящий для целей динамической трассировки.

Элементы языка C

Vue поддерживает подмножество языка C.

В следующей таблице показывается степень поддержки компилятором ProbeVue конкретных ключевых слов языка C. Все ключевые слова языка C остаются зарезервированными в языке Vue. Их использование в качестве имен переменных или других символов не является синтаксической ошибкой, однако последствия такого использования непредсказуемы.

Примечание: Ключевые слова во втором столбце могут употребляться в определениях типов структур и объявлениях функций. Компилятор Vue их не учитывает. Но использовать в объявлениях переменных сценария Vue эти ключевые слова нельзя.

Поддерживается	Разрешено в заголовочных файлах или в секции объявлений.	Не поддерживается
char	auto	break
double	const	case
else	extern	continue
enum	register	по умолчанию
float	static	do
if	typedef	for
int	volatile	goto
long		switch
return		while
short		
signed		
sizeof		
struct		
union		
unsigned		
void		

Список поддерживаемых Vue элементов языка C:

Операторы
Все операторы C, кроме цикла for и некоторых операторов управления ходом вычислений.

Операторы
Все унарные, бинарные и тернарные операторы C, кроме оператора "запятая". Приоритеты и ассоциативность операторов такие же, как в C.

Типы данных

Большая часть типов переменных C-89, включая все операторы и ключевые слова (struct, union, enum, typedef и т. д.) для объявления типов, поддерживается с ограничениями. Сюда входят типы для параметров и переменных приложения и ядра.

Примечание: В vue действуют свои правила областей видимости и классов памяти.

Преобразования типов

Неявные преобразования типов, а также явные с использованием операторов приведения типов.

Функция

Синтаксис вызова функций и передачи параметров в них. Обратите внимание на ограничения, связанные с вызовом функций.

Имена переменных

Требования к именам переменных соответствуют требованиям к идентификаторам языка C. Полная спецификация переменной может включать двоеточия, если перед именем переменной ставится префикс имени класса.

Файлы заголовков

Заголовочные файлы добавляются для явного объявления типов глобальных переменных ядра или прототипов функций ядра и приложений. Существуют определенные ограничения на порядок добавления заголовочных файлов.

Знаки препинания

Поддерживаются все знаки препинания языка C, правила их использования обязательны к выполнению. Например, операторы должны разделяться символом точки с запятой (**;**). Действуют все правила языка C касательно пробельных символов.

Литералы

Представление строк (с помощью двойных кавычек ("")), литералов символов (с помощью одинарных кавычек ('')), восьмеричных и шестнадцатеричных целых чисел, а также специальных символов (**\n**, **\t** и т. п.).

Комментарий

Поддерживаются комментарии в стиле C и C++. Комментарии могут находиться и внутри и снаружи блока. Строки, начинающиеся с символа #, игнорируются. Использовать этот символ для комментариев не рекомендуется.

Отличия от языка C

Поведение некоторых элементов языка Vue отличается от аналогов в языке C. Некоторые ограничения наложены ради эффективности, другие для обеспечения безопасного выполнения сценария Vue в режиме ядра и защиты тестируемого процесса.

Операторы циклов

В сценариях Vue запрещены циклы. Эта мера предосторожности предотвращает бесконечное выполнение тестов Vue.

Условные операторы

В сценариях Vue разрешены только операторы "if-else". Правильное использование операторов "if" позволяет выразить почти любое условие. Более эффективное средство описания высокоуровневых условий, доступное в сценариях Vue, — это предикаты.

Оператор return

Оператор return используется в Vue для немедленного прекращения выполнения блока действий. Оператор return в Vue не имеет параметров, так как блок действий Vue не возвращает значений.

Функции

Сценарии Vue не имеют доступа к функциям системы AIX и пользовательским библиотекам. Нет

поддержки создания архивов (библиотек функций) и пользовательских функций, вызываемых из тестов. Вместо них имеется встроенная библиотека функций, которые могут потребоваться программам динамического тестирования.

Вещественные числа
Переменные вещественных типов нельзя использовать в блоках точки тестирования ядра. Допускается их применение только в простых операторах присваивания и в качестве параметров функций Vue для вывода данных. Поддержка вещественных переменных языком Vue ограничивается считыванием их значений.

Изменения переменных
Внешние переменные не могут стоять в левой части оператора присваивания, то есть их нельзя изменять в сценарии Vue.

Файлы заголовков
Язык Vue не поддерживает явное включение заголовочных файлов в сценарий Vue. Имя включаемого заголовочного файла должно передаваться в параметрах команды **probevue**. Все операторы и директивы препроцессора C в заголовочных файлах игнорируются. Это может привести к непредсказуемому поведению. Для того чтобы избежать этого, следует создать заголовочный файл вручную или предварительно обработать набор заголовочных файлов препроцессором C и передать результат обработки в команду probevue. Прототипы функций и определения структур (объединений) могут содержаться в самом сценарии Vue. Они должны помещаться в самое начало сценария перед блоками тестов.

Препроцессор C
Операторы, определения макросов, директивы line и pragma и имена предопределенных макросов препроцессора C игнорируются.

Операции с указателями
Vue не поддерживает указатели на переменные сценария. Например, нельзя получить адрес переменной сценария. Однако, адреса переменных ядра можно брать и сохранять в переменных указательного типа языка Vue. Для таких переменных поддерживаются операции с указателями.

Прочее
- Триграфы не поддерживаются.
- Оператор запятая не поддерживается.
- В объявлениях переменных не поддерживается инициализация.

Сценарии Vue
В отличие от процедур в процедурных языках программирования блок действий в Vue не имеет выходных параметров и не возвращает значений.

В отличие от процедур в процедурных языках программирования блок действий в Vue не имеет выходных параметров и не возвращает значений. Он также не поддерживает наборы входных параметров. С другой стороны, в точке входа в тест операциям блока действий доступны данные контекста. Например, в блоке действий Vue можно ссылаться на параметры, переданные в функцию; если точка тестирования является точкой входа в функцию.

Предикаты

Когда блоки точек тестирования должны выполняться в зависимости от определенных условий, следует использовать предикаты. Секция предиката определяется наличием ключевого слова **когда** сразу после секции определения теста. Предикат содержит обычное условное выражение стиля C в круглых скобках.

Формат предиката следующий:

```
when ( <условие> )
```

Например:

```
when ( __pid == 1678 )
```

Пример сценария Vue

Пример сценария Vue:

```
/* Глобальные переменные автоматически инициализируются нулем */ [1]

 int count; /[2]
 /*
  * Файл: count.e
  *
  * Подсчет числа системных вызовов read и write,
  * сделанных процессом с ИД 400
  */

 @@BEGIN
 {
  printf("Запуск теста\n");
 }

 @@syscall:*:read:entry, @@syscall:*:write:entry [3]
  when (__pid == 400)[4]

 {[5]

  count++;
  /* Вывод сообщения через каждые 20 системных вызовов */
  if (count % 20 == 0)
   printf("Число вызовов read/write: %d\n", count);
  /* Выход, когда число системных вызовов превысит 100 */
  if (count > 100)
   exit();
 } [6]

 /* Вывод статистики перед выходом */
 @@END
 {
  printf("Завершение теста после %d системных вызовов.\n", count);
 }
```

Верхние индексы обозначают различные элементы в примере сценария Vue:

1. Комментарий
2. (Необязательный) Секция определений
3. Спецификация теста
4. (Необязательный) Предикат
5. Начало блока действий
6. Конец блока действий

Сценарий запускается следующей командой. Примечание: показанный вывод сценария — это только пример.

```
# probevue count.e
 Число вызовов read/writes: 20
 Число вызовов read/writes: 40
 Число вызовов read/writes: 60
 ...
 ...
```

Для запуска **probevue** требуются соответствующие права доступа. Необходимо войти в систему под учетной записью администратора или получить права на тестирование системных вызовов всех процессов системы.

Спецификация точки тестирования

Спецификация точки тестирования состоит из одного или нескольких кортежей точек тестирования.

Каждый кортеж определяет место кода, чье выполнение, или событие, чье наступление должно вызвать выполнение операций теста. С одним набором операций и предикатом (если имеется) можно связать несколько точек тестирования, перечислив через запятую кортежи точек тестирования вверху блока Vue.

Ниже перечислены некоторые из поддерживаемых типов датчиков:

- Тесты входа в пользовательскую функцию (тесты **uft**)
- Тесты входа в системный вызов или выхода из системного вызова (тесты **syscall**)
- Тесты, срабатывающие через определенный интервал времени (**интервальные** тесты)

Полный список поддерживаемых типов тестов приведен в разделе администратора тестов.

Кортеж точки тестирования представляет собой упорядоченный список полей, разделенных двоеточиями, которые уникально идентифицируют точку тестирования. Ниже показан общий формат, хотя обычно указывается только поле расположения, если точка тестирования является расположением теста.

```
@@ <probetype>:
<поля, разделенные двоеточиями (определяются типом теста)>:<расположение>
```

Допустимые значения полей в кортеже и длина кортежа определяются диспетчером тестов. Общие правила определения кортежей точек тестирования для диспетчеров тестов:

- Каждый кортеж точки тестирования содержит не менее 3-х полей.
- Первое поле всегда является идентификатором типа теста, то есть диспетчера тестов.
- Для диспетчеров тестов, поддерживающих трассировку процессов, второе поле содержит ИД процесса.
- Для диспетчеров тестов, поддерживающих тесты входа или выхода из функции, поле расположения (последнее поле) должно содержать ключевое слово **entry** или**exit** соответственно.
- Поля разделяются двоеточием (:).
- Символ звездочки в поле "*" кортежа точки тестирования обозначает все возможные значения этого поля. Например, диспетчер тестов **syscall** разрешает тестирование системных вызовов конкретного процесса или всех процессов. В первом случае второе поле должно содержать ИД тестируемого процесса. В последнем — *. Другое применение символа звездочки — сохранение двоичной совместимости с существующими сценариями после усовершенствования тестов в будущем. Например, диспетчер тестов **uft** в данный момент требует ставить в третьем поле звездочку. В будущем может быть введена поддержка имени модуля для ограничения тестирования только функциями конкретного модуля. И это имя будет указываться в третьем поле.
- Максимальная длина спецификации теста составляет 1023 символа.

Пример:

@@uft:34568:*:foo:entry
: Тест на входе в функцию **foo** процесса с ИД = 34568. Звездочка в третьем поле показывает, что тестируются функции **foo** всех модулей процесса.

@@syscall:*:read:exit
: Тест на выходе системного вызова **read**. Звездочка показывает, что системный вызов **read** тестируется у всех процессов.

@@interval:*:clock:500
: Тест срабатывает каждые 500 мс (реального времени). Звездочка здесь — заполнитель поля, зарезервированного на будущее.

ИД процесса часто неизвестен во время написания сценария Vue. Vue предоставляет простой метод, избавляющий от необходимости указывать конкретный ИД процесса во втором поле спецификации теста и в любом другом месте кода сценария Vue (например, в разделе предиката).

В одном сценарии Vue могут содержаться точки тестирования многих процессов в пространстве пользователя и пространстве ядра. Данные трассировки всегда выводятся в хронологическом порядке.

В дополнение к обычным точкам тестирования, определяемым диспетчерами тестов, Vue поддерживает две специальные точки тестирования. Каждый сценарий Vue может содержать точку тестирования @@BEGIN, где указываются действия, выполняемые перед включением тестов, и точку тестирования @@END, содержащую действия, выполняемые после завершения трассировки.

Блок действий

Блок действий содержит операции трассировки, выполняемые при срабатывании точки трассировки. Поддерживаются не только операции сбора и форматирования данных трассировки, можно использовать любые средства языка Vue.

Блок действий в Vue аналогичен процедуре в процедурных языках программирования. Он состоит из последовательности операторов, выполняемых в заданном порядке. Выполнение операторов в целом последовательное. Исключение составляют условный оператор "if-else" и оператор выхода из блока "return". Vue также поддерживает функцию **exit**, которая прерывает выполнение всего сценария и завершает сеанс трассировки. Vue не поддерживает операторы "for", "do" и "goto" языка C.

В отличие от процедур в процедурных языках программирования блок действий в Vue не имеет выходных параметров и не возвращает значений. Он также не поддерживает наборы входных параметров. С другой стороны, в точке входа в тест операциям блока действий доступны данные контекста. Например, в блоке действий Vue можно ссылаться на параметры, переданные в функцию; если точка тестирования является точкой входа в функцию.

Предикаты

Когда блоки точек тестирования должны выполняться в зависимости от определенных условий, следует использовать предикаты. Секция предиката определяется наличием ключевого слова **когда** сразу после секции определения теста. Предикат содержит обычное условное выражение стиля C в круглых скобках.

Формат предиката следующий:

```
when ( <условие> )
```

Например:

```
when ( __pid == 1678 )
```

Переменные ProveVue

Язык Vue поддерживает большинство традиционных типов данных C, то есть тех, которые распознаются спецификацией C-89. Кроме того, Vue включает в себя некоторые расширения, которые облегчают запись мощных программ динамической трассировки.

Vue поддерживает переменные трех различных областей действия:

- Переменные, которые локальны только для одного блока действия
- Переменные с глобальной областью действия
- Переменные с областью действия, локальной для нити

Кроме того, Vue имеет доступ к переменным со внешней областью действия, таким как глобальные переменные в ядре или пользовательские данные в тестируемом приложении.

В общем случае, переменные должны быть объявлены перед их первым использованием в сценарии, хотя Vue также поддерживает очень ограниченную форму неявного распознавания типа. Операторы объявления переменных внутри блока действия должны находиться перед любыми исполняемыми операторами. Они не

могут находиться внутри вложенных блоков, например, внутри оператора if. В некоторых случаях можно объявить переменные вне всех блоков действия, но тогда все объявления должны находиться перед первым блоком действия.

Классы переменных

Vue поддерживает несколько классов переменных с различными правилами их области действия, способа их инициализации, возможности их обновления и способа определения их типов. Как и в языке C, все операторы объявления для переменной должны предшествовать их первому использованию в сценарии.

Vue предоставляет специальные спецификаторы типа, которые добавляются к оператору объявления для индикации классов объявляемых переменных. Например, ключевое слово **__global** - это спецификатор класса, который можно включить в оператор объявления, чтобы указать, что объявляемые переменные принадлежат классу "global".

В следующем примере **foo** и **bar** объявлены как переменные глобального класса:

```
__global int foo, bar;
```

Vue также поддерживает неявное распознавание типа переменной на основании ее первого использования в сценарии. В этом случае не существует оператора объявления, но класс переменной по-прежнему может быть предоставлен прямым добавлением спецификатора класса к переменной в ее первой текстовой ссылке в сценарии:

```
global:count = 5; /* Первая ссылка на переменную count в сценарии */
```

В предыдущем примере ключевое слово **global:** является спецификатором, который указывает, что переменная **count** - это переменная глобального класса. Этой переменной будет также неявно присвоен тип int, потому что первая ссылка на нее является выражением присваивания, правая сторона которого - это целая константа.

Примечание: При указании спецификатора класса в операторе объявления необходимо использовать ключевое слово **__global**, а при объявлении переменной во время ее первого использования в сценарии применяется ключевое слово **global:**. Правила синтаксиса подобны другим спецификаторам классов, поддерживаемым в Vue.

Переменные автоматического класса

Автоматические переменные специфичны для оператора и подобны автоматическим переменным или переменным стека в C. Их область находится внутри группы действий блока, в котором они определены, или они создаются при каждом вызове. Автоматические переменные всегда неопределенны в начале блока действия и должны быть инициализированы с помощью оператора присваивания перед их использованием в выражении или в другом исполняемом операторе.

Автоматическая переменная определяется с помощью префикса **auto:**, например, `auto:lticks` указывает на автоматическую переменную. Также можно объявлять автоматические переменные с помощью оператора объявления **__auto**. В этом случае приставку **auto:** можно опустить.

Нельзя использовать переменные автоматического класса в разделе предиката блока оператора Vue.

Следующий сценарий является примером оператора объявления **__auto**:

```
__auto int i;      /* Явное объявление */
auto:j = 0;        /* Неявное объявление */
```

Переменные класса локальной нити

Экземпляр переменной локальной нити создается в трассируемой нити в момент первого выполнения группы действий, которая присваивает значение этой переменной. После создания переменной локальной нити она существует, пока активен сценарий Vue и существует трассируемая нить. Значение переменной локальной нити связано с нитью и сохраняется при выполнении других блоков этой программы. Другими

словами, переменные этого класса видимы повсюду в сценарии Vue. Однако, каждая нить, которая выполняет сценарий Vue, получает свои собственные копии этих переменных, и переменные в каждой такой копии доступны и изменяемы повсюду в сценарии, но только для нити, которая создала их экземпляр.

Переменная локальной нити имеет приставку **thread:**. Например, `thread:count`. Также можно объявлять переменные локальной нити с помощью оператора объявления **__thread**. В этом случае приставку **thread:** можно опустить.

Можно использовать переменную локальной нити в разделе предиката оператора Vue даже до того, как создан ее экземпляр. Предикат с переменной локальной нити, экземпляр которой не создан, всегда имеет значение FALSE. При использовании в разделе предиката должен быть всегда включен префикс **thread:**, чтобы указать на переменную локальной нити.

Следующий сценарий является примером оператора объявления **__thread**:

```
__thread int i;      /* Явное объявление */
thread:j = 0;        /* Неявное объявление */
```

Примечание: Хотя можно объявить переменные локальной нити внутри тестов **@@BEGIN** и **@@END**, все другие ссылки на них в этих специальных тестах могут вызвать неопределенное поведение. Сам оператор объявления не вызывает создания экземпляра переменной локальной нити.

Переменные глобального класса

Переменные глобального класса имеют глобальную область действия и видимы повсюду в сценарии Vue. Можно использовать глобальные переменные в одном или нескольких операторах сценария Vue. Они также могут быть объявлены для ясности в начале перед первым оператором. Глобальные переменные инициализируются значением ноль или пустым значением.

Всем переменным в сценарии Vue по умолчанию присваивается глобальный класс, если явно не добавлен в виде префикса спецификатор не глобального класса. Можно также явно объявить глобальные переменные с помощью спецификатора класса **__global** при объявлении переменной. Переменные списка по определению всегда создаются как переменные глобального класса.

Чтения и изменения глобальных переменных не сериализуются, если они не имеют тип списка. Нет гарантии сбора данных, когда тесты выполняются одновременно. Глобальные переменные, которые имеют тип, отличный от списка, полезны для сбора профилирования и других статистических данных.

Можно использовать глобальные переменные в разделе предиката оператора Vue.

Следующие сценарии являются примерами инициализации и использования глобальных переменных:

```
int wcount;   /* Глобальная переменная объявлена перед первым оператором */

 @@BEGIN
 {
  int f_count;  /* Глобальная переменная объявлена  внутри @@BEGIN */
  __global int z_count; /* Глобальная переменная объявлена с префиксом __global */

  f_count = 12;
 }

 @@syscall:*:read:entry
  when (z_count == 0)
 {
  int m_count;  /* Глобальная переменная объявлена внутри теста */
  m_count += f_count; /* f_count уже объявлена в более раннем тесте */
  printf("m_count = %d\n", m_count);
  if (wcount == 1)
   exit();
 }
```

```
@@syscall:*:write:entry
{
 m_count++;  /* m_count уже объявлена в более раннем тесте */

}

@@syscall:*:write:exit
{
 wcount = 1;  /* w_count объявлена глобально */
}
```

Переменные глобального класса ядра

В ProbeVue администратор имеет доступ к глобальным переменным ядра внутри блока действия любого оператора Vue даже для точек теста в пользовательском пространстве, таком как точки теста uft. Перед использованием или ссылкой на переменную ядра в сценарии Vue необходимо явно объявить ее с помощью оператора объявления **__kernel**. Доступны только переменные, экспортированные ядром, то есть те, которые присутствуют в списке экспорта **/unix**.

Переменные ядра не могут использоваться в разделе предиката блока. Переменные ядра всегда считаются переменными только для чтения в сценарии Vue. Любая попытка записи в переменную ядра или вызовет синтаксическую ошибку, или приведет к аварийному прекращению сценария.

Пример объявления и использования переменных ядра в сценарии Vue.

Обращайтесь только к закрепленным переменным. Если страница, содержащая переменную ядра, находится не в памяти (вытеснена), ProbeVue возвращает нулевое значение для этой переменной.

Можно получить доступ к перемененным ядра интегрального типа и переменным ядра, которые являются структурами или объединениями и даже указателями. Далее, можно также ссылаться на имена структур ядра и объединений в сценарии Vue. Есть также доступ к массивам ядра, но не существует поддержки копирования символьных данных ядра в строку ProbeVue.

Полезные переменные ядра

В следующей таблице перечислены несколько примеров полезных переменных ядра, к которым можно обратиться из сценария Vue. Будьте осторожны при использовании их в сценарии Vue, так как имена этих переменных или их значения могут изменяться в разных выпусках AIX. Все эти переменные ядра закреплены в памяти и экспортированы из ядра.

Переменная ядра	Описание	Связанные файлы заголовка
struct **system_configuration _system_configuration**	Определение конфигурации системы.	sys/systemcfg.h
struct var **v**	Основные настраиваемые параметры ядра (и другие).	sys/var.h
struct timestruc_t **tod**	Время суток, указанное в памяти. Секунды и наносекунды от начала периода.	sys/time.h
cpu_t **high_cpuid**	Самый высокий ИД CPU.	sys/encap.h
struct vminfo **vmminfo**	Структура данных, которая содержит информацию, показанную командой **vmstat**.	sys/vminfo.h
time_t **lbolt**	Количество тактов после последней загрузки.	sys/time.h
char **spurr_version**	Определяет, поддерживает ли текущая система регистр SPURR: 0=No SPURR, 1=CPUs have SPURR.	sys/sysinfo.h
struct **utsname utsname**	Структура системного имя, которая включает в себя название операционной системы, имя узла, уровень выпуска и так далее.	sys/utsname.h

Переменные класса входа

Блоки, связанные с точками тестирования, которые находятся в точке расположения входа системного вызова или пользовательской функции, могут обращаться к аргументам, переданным тестируемой функции или системному вызову.

Тесты во входных точках поддерживаются системными вызовами и пользовательской функцией диспетчера трассировки тестирования. Например, системный вызов read имеет три аргумента: ИД файла описания, указатель на буфер пользователя и значение числа считываемых байтов. К этим значениям можно обращаться, если спецификацией теста - это **@@syscall:*:read:entry**, который является точкой входа системного вызова.

Параметры функции указываются с помощью специальных имен встроенных переменных: **__arg1**, **__arg2**, **__arg3** - вплоть до числа аргументов, переданных функции. Например, в операторе, связанном с точкой входа системного вызова read, **__arg1** ссылается на значение параметра ИД файла описания, **__arg2** ссылается на значение параметра указателя на буфер, а **__arg3** на размер считываемых данных.

Примечание: Когда указана одна или несколько записей точки теста, переменные __arg <x> не разрешены в блоке действия и приведут к ошибке, как показано в следующем примере.

```
@@syscall:*:read:entry,@@syscall:*:write:entry
{
        char *argument;
        argument=__arg2;  -> Не разрешено.
}
```

Probevue вызовет следующее сообщение об ошибке: *Невозможно использовать встроенный аргумент. Функция не определена.*

Использование переменных класса входа в операторе Vue разрешено, только если объявление в стиле C тестируемой функции и типа данных передаваемых в нее параметров также предоставлено в сценарии Vue. Оно должно находиться перед первым оператором Vue, который ссылается на оператор входа. Поместите объявление перед операторами Vue в начале сценария Vue.

Следующий сценарий является примером использования переменных класса входа:

```
int read(int fd, char *buf, unsigned long size);

@@syscall:*:read:entry
{
 printf("Number of bytes to read = %d\n", __arg3);
}
```

Примечание: В приведенном выше примере определение функции системного вызова read, указанное в сценарии, не соответствует точно тому, что дано в файле **/usr/include/unistd.h**, но это также работает.

Второе требование состоит в том, чтобы спецификация теста, связанная с оператором, определяла уникальную точку теста. Переменные класса входа не могут быть использованы в операторе Vue, который имеет несколько точек теста, указанных в спецификации теста, независимо от того, является ли тестируемая функция той же самой или имеет подобные прототипы функций. Следующий сценарий является недопустимым и вызовет синтаксическую ошибку в компиляторе ProbeVue, так как спецификация теста включает в себя две точки теста:

```
int read(int fd, char *buf, unsigned long size);
int write(int fd, char *buf, unsigned long size);

@@syscall:*:read:entry, @@syscall:*:write:entry
{
 /* Здесь невозможно использовать __arg3, так как этот оператор имеет несколько точек теста,
  * связанных с ним. Этот точка вызовет синтаксическую
```

```
   * ошибку при компиляции команды probevue.
   */
 printf("Number of bytes to read/write = %d\n", __arg3);
}
```

Следующий измененный сценарий работоспособен:

```
int read(int fd, char *buf, unsigned long size);
int write(int fd, char *buf, unsigned long size);

@@syscall:*:read:entry
{
 printf("Number of bytes to read = %d\n", __arg3);
}
@@syscall:*:write:entry
{
 printf("Number of bytes to write = %d\n", __arg3);
}
```

Переменные класса выхода

Блоки, связанные с точками тестирования, которые находятся в точке расположения выхода системного вызова или пользовательской функции, могут обращаться к значению возврата пользовательской функции или системного вызова.

Существует только одна переменная класса выхода, определенная в языке Vue. Это значение возврата из функции или системного вызова, доступ к которому можно получить с помощью имени специальной встроенной переменной **__rv**.

Тесты во выходных точках поддерживаются системными вызовами диспетчера трассировки тестирования. Например, системный вызов read возвращает фактическое число прочитанных байт или код возврата ошибки -1. К данному значению возврата можно обратиться в точке тестирования **@@syscall:*:read:exit**, которая определяет все точки выхода из системного вызова read.

Подобно переменным класса входа, использование переменных класса выхода в операторе Vue допустимо, только если спецификация теста, связанная с оператором, определяет уникальную точку теста. Таким образом, **__rv** не может быть использована в операторе Vue, который имеет несколько точек теста, указанных в спецификации теста. Кроме того, объявление в стиле C тестируемой функции и типа данных значения возврата должно быть явно предоставлено в сценарии Vue. Фактически, ошибкой является указание объявления функции без предоставления ее типа возврата.

Можно использовать переменные класса выхода в разделе предиката блока.

Следующий сценарий является недопустимым и вызовет синтаксическую ошибку в компиляторе ProbeVue, так как не задан тип функции **read**:

```
/* Неправильный пример.  */

 int read(int fd, char *buf, unsigned long size);

 @@syscall:*:read:exit
  when (__rv > 0)
 {
  /* Введено при успешном чтении: значение возврата = # числу прочитанных байтов */
  printf("Number of bytes read = %d\n", __rv);
 }
```

Следующий измененный сценарий работоспособен:

```
/* Правильный пример.  */

 int read(int fd, char *buf, unsigned long size);
```

```
@@syscall:*:read:exit
 when (__rv > 0)
{
 /* Введено при успешном чтении: значение возврата = # числу прочитанных байтов */
 printf("Number of bytes read = %d\n", __rv);
}
```

Встроенные переменные класса

В дополнение к специальным встроенным переменным, от **__arg1** до **__arg32** и **__rv**, Vue также определяет набор встроенных переменных общего назначения. Встроенные переменные общего значения рассматриваются в этом разделе, а встроенные переменные администратора тестов - в разделе, посвященному администратору тестов. Встроенные переменные являются функциями, но ProbeVue считает их переменным. Поэтому эти встроенные переменные можно использовать в разделе предиката блока языка Vue.

В Vue поддерживаются следующие встроенные переменные:

__tid
ИД трассируемой нити.

__pid
ИД процесса трассирующей нити.

__ppid
ИД родительского процесса трассируемой нити.

__pgid
ИД группы процессов трассируемой нити.

__pname
Имя процесса трассирующей нити.

__uid, __euid
Реальный и эффективный ИД пользователя трассируемой нити.

__trcid
ID трассируемого процесса (то есть, команды **probevue**)

__errno
Текущее значение errno для трассируемой нити.

__kernelmode
Текущий исполняемый режим: или 1 (режим ядра), или 0 (режим пользователя).

__r3, ..., __r10
Значения регистров общего назначения (для параметров функции или значений возврата).

__curthread
Текущая нить.

__curproc
Текущий процесс.

__ublock
Область пользователя текущего процесса.

__mst
Встроенная переменная для доступа к содержимому аппаратного регистра области, в которой сохраняется состояние текущей нити (MST).

__stat
Встроенная переменная для доступа к статистике системы для разных компонентов ядра AIX®.

Следующий сценарий является примером использования встроенных переменных:

```
@@syscall:*:read:entry
{
 printf("Thread ID:%d, Process ID:%d, Parent Process ID:%d\n",
   __tid, __pid, __ppid);
 printf("Process Group ID: %d\n", __pgid);
 printf("Process name = %s\n", __pname);

 printf("Real UID=%d, Effective UID=%d\n", __uid, __euid);S

 printf("probevue command process ID = %d\n", __trcid);

 printf("Errno = %d\n", __errno);
 printf("Mode = %s\n", __kernelmode == 1 ? "kernel" : "user");

 printf("Current values of GPRs: r3=0x%016llx, r4=0x%016llx, r5=0x%016llx\n",
   __r3, __r4, __r5);
 printf("                        r6=0x%016llx, r7=0x%016llx, r8=0x%016llx\n",
   __r6, __r7, __r8);
 printf("                        r9=0x%016llx, r10=0x%016llx\n",
   __r9, __r10);
}
```

Встроенная переменная __curthread:

__curthread - это встроенная переменная, с помощью которой пользователь может получить сведения о текущей нити. Доступ к этим сведениям можно получить, применив оператор **-->** к переменной **__curthread**. Данную переменную нельзя использовать в тестах **systrace**, **BEGIN** и **END**. Допускается использование этой переменной в тестах с интервалами если указан **PID**. Данная переменная предоставляет функциональность, схожую с **getthrds/getthrds64**, но ограниченную текущей нитью. Доступные данные:

tid
ИД нити

pid
Идентификатор процесса

стратегия
Стратегия планирования

pri
Приоритет

cpusage
Использование CPU

cpuid
Процессор, на котором выполняется данная нить

sigmask
Сигнал, заблокированный в нити

lockcount
Число блокировок ядра нитью

ptid
Идентификатор pthread для этой нити (0, если нить ядра; 1, если приложение с одной нитью)

homecpu
Домашний CPU нити.

homesrad
Домашний srad нити

Пример использования

K Tid текущей нити можно обратиться с помощью __curthread->tid.

Встроенная переменная __curproc:

__curproc - это встроенная переменная, с помощью которой пользователь может получить сведения о текущем процессе. Доступ к этим сведениям можно получить, применив оператор **->** к переменной **__curproc**. Данную переменную нельзя использовать в тестах **systrace**, **BEGIN** и **END**. Допускается использование этой переменной в интервальных тестах если указан **PID**. Данная переменная предоставляет функциональность, схожую с **getproc**, но ограниченную текущим процессом. Доступные данные:

pid
ИД процесса.

ppid
ИД родительского процесса

pgid
ИД группы процессов

uid
Фактический ИД пользователя

suid
Сохраненный ИД пользователя

pri
Приоритет

nice
значения nice

cpu
Использование процессора

adspace
Адресное пространство процесса

majflt
Страничная ошибка ввода-вывода

minflt
Ошибка ввода-вывода, не являющаяся страничной

size
Размер образа в страницах

sigpend
Ожидающие сигналы процесса

sigignore
Сигналы, игнорированные процессом

sigcatch
Сигналы, обработанные процессом

forktime
Время создания процесса

threadcount
Число нитей в процессе

cwd
Текущий рабочий каталог. Если контекст страничной ошибки недоступен, или размер стека вычисления на CPU меньше, чем 96 Кб, или страничные ошибки не обрабатываются в тесте (например тест с интервалами), то эта встроенная функция возвращает строку, равную null

Пример использования

Для обращения к идентификатору родительского процесса текущего процесса можно ввести `__curproc->ppid`.

Встроенная переменная __ublock:

__ublock - это встроенная переменная, с помощью которой пользователь может получить сведения о текущем процессе. Данную переменную нельзя использовать в тестах **systrace**, **BEGIN** и **END**. Допускается использование этой переменной в тестах с интервалами если указан **PID**. Доступ к этим сведениям можно получить, применив оператор **-->** к переменной **__ublock**. Доступные данные:

text
Начало текста

tsize
Размер текста (в байтах)

data
Начало данных

sdata
Текущий размер данных (в байтах)

mdata
Максимальный размер данных (в байтах)

stack
Начало стека

stkmax
Максимальный размер стека (в байтах)

euid
Действующий ИД пользователя

uid
Фактический ИД пользователя

egid
Действующий ИД группы

gid
Фактический ИД группы

utime
Время использования ресурсов пользователя процессом (в секундах)

stime
Время использования системных ресурсов процессом (в секундах)

maxfd
Максимальное значение fd пользователя

is64u
1, если в контексте 64-разрядного процесса

Пример использования

Для обращения к началу текста текущего процесса можно ввести `__ublock->text`.

Встроенная переменная __mst:

__mst - это специальная встроенная переменная, предоставляющая доступ к аппаратным регистрам текущей нити. Данную переменную нельзя использовать в **systrace**, **BEGIN** и **END**. Допускается использование этой переменной в интервальных тестах, если указан **PID**. Доступ к этим сведениям можно получить, применив оператор **-->** к переменной **__ublock**. Можно получить доступ к следующим регистрам:

r1-r10
Регистры общего назначения от r1 до r10

r14-r31
Регистры общего назначения от r14 до r31

iar
Регистр адреса инструкции

lr Регистр ссылки

isisr
Задано, если в контексте исключительной ситуации или прерывания.

Пример использования

Для получения доступа к **lr** в тесте используйте команду:

```
__mst->lr
```

Встроенная переменная __stat:

Эта встроенная переменная обеспечивает доступ к статистике системы для разных компонентов ядра AIX с помощью сценария Vue. Статистика системы доступна в виде счетчиков, к которым можно обратиться из любой точки теста в любом сценарии ProbeVue. Новая точка теста не добавляется для поддержки статистики системы. Для доступа к статистике системы требуются права на трассировку ядра `aix.ras.probevue.trace`.

Преимущества статистики системы:

- Для доступа к статистике не требуется включать трассировку системы или трассировку компонентов.
- Отображаются только обязательные поля. Это невозможно в случае применения текущих команд статистики. Прямой доступ к полям из структур ядра позволяет избежать копирования больших объемов данных.
- Статистика системы теперь доступна в сценариях Vue для выполнения арифметических логических операций. Например, с помощью ProbeVue можно добавить операции для двух дисков.

Встроенная переменная Vue **__stat** позволяет извлекать статистику как на глобальном уровне, так и на уровне отдельных компонентов. Данные предоставляются в виде счетчиков, к которым периодически обращаются сценарии Vue. Сохранив требуемые значения, можно вычислить конечные результаты. К статистике системы можно обращаться в следующих целях:

- Создание простого инструмента сбора статистики с помощью сценария Vue без вызова API C/C++ для печати разностных значений счетчиков каждую секунду или с настраиваемым интервалом.
  - Отслеживание разностного значения счетчика с учетом порогового значения. В случае превышения порогового значения в протокол заносится сообщение.
  - С помощью сценария Vue можно отслеживать фактическое значение счетчика с учетом порогового значения (например, максимальное время обслуживания диска).

Встроенная переменная **__stat** предлагает несколько режимов доступа к статистике из источника. Источник статистики системы можно получить из разных компонентов системы. Ниже перечислены режимы доступа к статистике.

Доступ в синхронном режиме
ProbeVue обеспечивает прямой доступ к статистике системы в ходе выполнения сценария Vue. Прямой доступ позволяет обращаться к текущим данным. По умолчанию ProbeVue выбирает режим прямого доступа (если он доступен). В этом режиме доступны не все данные статистики, поскольку прямой доступ может не поддерживаться компонентами или текущей нитью.

Асинхронный или кэшируемый режим
Данные периодически собираются в источнике и кэшируются в ProbeVue. Сценарий Vue обращается

к данным с помощью кэша. Интервал обновления кэша можно настроить на уровне отдельного сеанса или всех сеансов. Каждый исходный компонент предлагает метод доступа к статистике в асинхронном режиме. В таких случаях можно предоставить доступ ко всей статистики в кэшируемом режиме с помощью параметра **fetch_stats_async_only**. Кэшируемый режим применяется, если текущие данные недоступны или не требуются.

Примеры

1. Следующий пример выдает число операций ввода-вывода для диска `hdisk9`, выполняемых каждую секунду:

```
@@interval:*:clock:1000
{
    printf("Number of transfers = %lld\n", __stat.io.disk.hdisk9->transfers);
}
```

2. В следующем примере время обслуживания измеряется в микросекундах:

```
@@syscall:*:read:exit
{
    rdservtime = __stat.io.disk.hdisk10->rd_service_time;
    printf("rdservtime=%lld microseconds\n", rdservtime);
}
```

Синтаксис встроенной переменной __stat

Общий синтаксис выражений со встроенными переменными **__stat**:
`__stat.<level1_keyword>[.<level2_keyword>.....][.<inst1_keyword>....]-><fieldname>`

В следующей таблице перечислены стандартные уровни и экземпляры для статистики ввода-вывода памяти:

Таблица 1. Стандартные уровни и экземпляры

Встроенный	Уровень 0	Уровень 1	Экземпляр 0	Экземпляр 1	Имена полей
__stat	io	disk	hdisk[0...n]		• Обратитесь к разделу Табл. 3 на стр. 217
			hdisk[0...n]	path[0...n]	• Обратитесь к разделу Табл. 4 на стр. 218
__stat	io	adapter	vscsi[0....n]		• Обратитесь к разделу Табл. 5 на стр. 218 • Обратитесь к разделу Табл. 6 на стр. 219
__stat	io	adapter	fcs[0...n]		• Обратитесь к разделу Табл. 7 на стр. 219

В следующей таблице перечислены стандартные уровни и экземпляры для сетевой статистики:

Таблица 2. Стандартные уровни и экземпляры

Встроенный	Уровень 0	Уровень 1	Уровень 2	Экземпляр 0	Имена полей
__stat	net	adapter		ent[0...n]	• Обратитесь к разделу Табл. 8 на стр. 221
__stat	net	interface		en[0....n]	• Обратитесь к разделу Табл. 9 на стр. 224
__stat	net	protocol	ip		• Обратитесь к разделу Табл. 10 на стр. 225
__stat	net	protocol	ipv6		• Обратитесь к разделу Табл. 11 на стр. 227
__stat	net	protocol	tcp		• Обратитесь к разделу Табл. 12 на стр. 229
__stat	net	protocol	udp		• Обратитесь к разделу Табл. 13 на стр. 232
__stat	net	protocol	icmp		• Обратитесь к разделу Табл. 14 на стр. 233
__stat	net	protocol	icmpv6		• Обратитесь к разделу Табл. 15 на стр. 233
__stat	net	protocol	igmp		• Обратитесь к разделу Табл. 16 на стр. 235
__stat	net	protocol	arp		• Обратитесь к разделу Табл. 17 на стр. 236

Статистика дискового ввода-вывода SCSI

В следующей таблице перечислены имена поддерживаемых полей для статистики дискового ввода-вывода SCSI (Small Computer System Interface). Эти поля относятся к экземплярам дисков. К ним можно обратиться с помощью `__stat.io.disk.<hdisk0...n>->имя-поля`. Следующую статистику можно получить как в синхронном, так и в асинхронном режиме.

Таблица 3. Статистика дискового ввода-вывода SCSI

Имя поля сценария Vue	Тип данных	Описание
name	String[32]	Имя диска
block_size	unsigned long long	Размер блока диска в байтах
transfers	unsigned long long	Число операций обмена данными с диском
rd_block_count	unsigned long long	Число прочитанных блоков диска
rd_service_time	unsigned long long	Общее время чтения или получения службы в микросекундах.
rd_min_service_time	unsigned long long	Минимальное время чтения службы в микросекундах.
rd_max_service_time	unsigned long long	Максимальное время чтения службы в микросекундах.
rd_timeouts	unsigned long long	Число тайм-аутов чтения
rd_failures	unsigned long long	Число ошибок чтения
wr_block_count	unsigned long long	Число записанных блоков
wr_service_time	unsigned long long	Общее время записи службы в микросекундах.
wr_min_service_time	unsigned long long	Минимальное время записи службы в микросекундах.
wr_max_service_time	unsigned long long	Максимальное время записи службы в микросекундах.
wr_timeouts	unsigned long long	Число тайм-аутов записи
wr_failures	unsigned long long	Число тайм-аутов записи
wait_queue_depth	unsigned long long	Глубина очереди ожидание драйвера
accum_wait_queue_time	unsigned long long	Общее время ожидания в очереди в микросекундах
min_wait_queue_time	unsigned long long	Минимальное время ожидания очереди в микросекундах.
max_wait_queue_time	unsigned long long	Максимальное время ожидания очереди в микросекундах.
num_queue_full	unsigned long long	Общее число обрабатываемых элементов в очереди

Статистика ввода-вывода дисковых путей SCSI

В следующей таблице перечислены имена поддерживаемых полей для статистики ввода-вывода дисковых путей SCSI (Small Computer System Interface). Эти поля относятся к экземплярам дисков и путей. К ним можно обратиться с помощью `__stat.io.disk.<hdisk0...n>.path[0...n]->имя-поля`. Следующую статистику можно получить как в синхронном, так и в асинхронном режиме.

Примечание: Статистика ввода-вывода дисковых путей SCSI поддерживает только драйвер альтернативных путей IBM® (MPIO).

Таблица 4. Статистика ввода-вывода дисковых путей SCSI

Имя поля сценария Vue	Тип данных	Описание
name	String[32]	Имя диска
block_size	unsigned long long	Размер блока диска в байтах
transfers	unsigned long long	Число операций обмена данными с диском
rd_block_count	unsigned long long	Число прочитанных блоков диска
wr_block_count	unsigned long long	Число записанных блоков

Статистика ввода-вывода клиента vSCSI

В следующей таблице перечислены имена поддерживаемых полей для статистики ввода-вывода клиента виртуального SCSI (vSCSI). Эти поля относятся к экземплярам клиентов vSCSI. К ним можно обратиться с помощью `__stat.io.adapter.<vscsi0...n>->имя-поля`. Следующую статистику можно получить как в синхронном, так и в асинхронном режиме.

Таблица 5. Статистика ввода-вывода клиента vSCSI

Имя поля сценария Vue	Тип данных	Описание
name	String[32]	Имя устройства
transfers	unsigned long long	Число операций обмена данными с устройством
rd_block_count	unsigned long long	Число прочитанных блоков
rd_service_time	unsigned long long	Общее время чтения или получения службы в микросекундах.
rd_min_service_time	unsigned long long	Минимальное время чтения службы в микросекундах.
rd_max_service_time	unsigned long long	Максимальное время записи службы в микросекундах.
wr_block_count	unsigned long long	Число записанных блоков
wr_service_time	unsigned long long	Общее время записи службы в микросекундах.
wr_min_service_time	unsigned long long	Минимальное время записи службы в микросекундах.
wr_max_service_time	unsigned long long	Максимальное время записи службы в микросекундах.
wait_queue_depth	unsigned long long	Глубина очереди ожидания для драйвера
accum_wait_queue_time	unsigned long long	Общее время ожидания в очереди в микросекундах.
min_wait_queue_time	unsigned long long	Минимальное время ожидания очереди в микросекундах.
max_wait_queue_time	unsigned long long	Максимальное время ожидания очереди в микросекундах.
num_queue_full	unsigned long long	Общее число обрабатываемых элементов в очереди

Статистика драйвера клиента vSCSI

В следующей таблице перечислены имена поддерживаемых полей для статистики драйвера клиента виртуального SCSI (vSCSI). Эти поля относятся к экземплярам клиентов vSCSI. К ним можно обратиться с помощью `_stat.io.adapter.<vscsi[0...n]>->имя-поля`. Следующую статистику можно получить как в синхронном, так и в асинхронном режиме.

Таблица 6. Статистика драйвера клиента vSCSI

Имя поля сценария Vue	Тип данных	Описание
no_dma_failures	unsigned char	Сколько раз системе не удалось отправить команду ввода-вывода из-за недостаточного размера области прямого доступа к памяти (DMA). Например, DMA_NORES
no_cmd_elem_failures	unsigned char	Сколько раз системе не удалось отправить команду ввода-вывода из-за отсутствия свободного элемента команды в драйвере клиента.
num_ping_timeouts	unsigned char	Число ошибок запросов проверки связи драйвера адаптера со связанным сервером виртуального ввода-вывода (VIOS).
num_bad_mad	unsigned char	Сколько раз системе не удалось обработать дейтаграмму управления, поскольку адаптер не находился в активном состоянии.
num_hcall_drops	unsigned char	Сколько раз системе не удалось отправить команду CRQ хоста (VIOS) из-за переполнения очереди ответов на команду (CRQ). Например, ошибки в случае применения параметров `H_SEND_CRQ()` и `H_DROPPED`.

Статистика драйвера Fiber Channel

В следующей таблице перечислены имена поддерживаемых полей для статистики драйвера Fiber Channel. Эти поля относятся к экземплярам устройств Fiber Channel. Синтаксис оператора Vue: `__stat.io.adapter.fcs[0...n]->имя-поля`. Следующая статистика доступна только в асинхронном режиме.

Таблица 7. Статистика драйвера Fiber Channel

Имя поля сценария Vue	Тип данных	Описание
secs_since_last_reset	unsigned long long	Время в секундах с момента последнего сброса
tx_frames	unsigned long long	Число переданных кадров
tx_words	unsigned long long	Объем переданных данных Fibre Channel в килобайтах
rx_frames	unsigned long long	Число полученных кадров
rx_words	unsigned long long	Объем полученных данных Fibre Channel в килобайтах
lip_count	unsigned long long	Число событий LIP в кольце с арбитражной логикой Fibre Channel (FC-AL)
nos_count	unsigned long long	Число событий NOS (Нет операционной системы)
error_frames	unsigned long long	Число кадров с ошибками контрольной суммы (CRC) или отброшенных кадров. Значение этого поля зависит от конкретного адаптера.
lost_frames	unsigned long long	Число потерянных кадров
link_fail_count	unsigned long long	Число сбоев связи
sync_loss_count	unsigned long long	Число ошибок, связанных с потерей синхронизации
sig_loss_count	unsigned long long	Число ошибок, связанных с потерей сигнала
prim_seq_proto_errcount	unsigned long long	Число простых ошибок порядка

Таблица 7. Статистика драйвера Fiber Channel (продолжение)

Имя поля сценария Vue	Тип данных	Описание
inval_words_received	unsigned long long	Число полученных недопустимых слов передачи
inval_crc_count	unsigned long long	Число ошибок CRC в полученных кадрах
num_interrupts	unsigned integer	Общее число прерываний
num_spurious_interrupts	unsigned integer	Общее число ложных прерываний.
elastic_buf_overrun_errcount	unsigned integer	Число переполнений эластичного буфера в интерфейсе линии связи.
in_reqs	unsigned long long	Входящие запросы
out_reqs	unsigned long long	Исходящие запросы
ctrl_reqs	unsigned long long	Управляющие запросы
in_bytes	unsigned long long	Входящие байты
out_bytes	unsigned long long	Исходящие байты
no_dma_resource_count	unsigned long long	Число сбоев DMA из-за недоступных ресурсов DMA
no_adap_elems_count	unsigned long long	Сколько раз не удалось выделить элемент команды адаптера из-за отсутствия доступных элементов
no_cmd_resource_count	unsigned long long	Сколько раз не удалось выделить команду из-за отсутствия доступных ресурсов
adap_num_active_cmds	unsigned integer	Число активных команд в драйвере адаптера
adap_active_high_wmark	unsigned integer	Максимальное число активных запросов в драйвере адаптера
adap_num_pending_cmds	unsigned integer	Число ожидающих команд в драйвере адаптера
adap_pending_high_wmark	unsigned integer	Максимальное число ожидающих запросов в драйвере адаптера
adap_heldoff_num_cmds	unsigned integer	Число команд в очереди задержки драйвера адаптера
adap_heldoff_high_wmark	unsigned integer	Максимальное число команд в очереди задержки драйвера адаптера
proto_num_active_cmds	unsigned integer	Число активных команд в драйвере SCSI-FC
proto_active_high_wmark	unsigned integer	Максимальное число активных запросов в драйвере SCSI-FC
proto_num_pending_cmds	unsigned integer	Число ожидающих команд в драйвере SCSI-FC
proto_pending_high_wmark	unsigned integer	Максимальное число ожидающих запросов в драйвере SCSI-FC

Статистика драйвера сетевого устройства

В следующей таблице перечислены имена поддерживаемых полей для статистики драйвера сетевого устройства. Эти поля относятся к экземплярам устройств сетевых устройств. К статистике драйвера сетевого устройства можно обратиться с помощью `"__stat.net.adapter.<ent0...n>->имя-поля`

Таблица 8. Статистика драйвера сетевого устройства

Имя поля сценария Vue	Тип данных	Описание	Тип доступа к полю (Асинхронный или Оба)
flags	unsigned int	Значения флагов адаптера. Возможные значения этого поля: • NDD_UP • NDD_BROADCAST • NDD_DEBUG • NDD_RUNNING • NDD_SIMPLEX • NDD_DEAD • NDD_LIMBO • NDD_PROMISC • NDD_ALTADDRS • NDD_MULTICAST • NDD_DETACHED • NDD_64BIT • NDD_HIGHFUNC_QOS • NDD_MEDFUNC_QOS • NDD_MINFUNC_QOS • NDD_QOS • NDD_CHECKSUM_OFFLOAD • NDD_PSEG • NDD_ETHERCHANNEL • NDD_VLAN • NDD_SPECFLAGS Эти значения доступны в виде символьных констант.	оба варианта
max_mtu	unsigned int	Максимальный блок передачи.	оба варианта
min_mtu	unsigned int	Минимальный блок передачи.	оба варианта
type	unsigned int	Типы интерфейсов. Возможные значения этого поля: • NDD_ETHER • NDD_ISO88023 • NDD_ISO88024 • NDD_ISO88025 • NDD_ISO88026 Эти значения доступны в виде символьных констант. **Примечание:** Поскольку заданы не все возможные значения типов интерфейса, значение может содержать другие опции.	оба варианта
physaddr	mac_addr_t	Физический адрес или MAC-адрес.	оба варианта

Таблица 8. Статистика драйвера сетевого устройства (продолжение)

Имя поля сценария Vue	Тип данных	Описание	Тип доступа к полю (Асинхронный или Оба)
adapter_type	unsigned int	Расширение поля флага. Возможные значения этого поля: • NDD_2_SEA • NDD_2_VIOENT • NDD_2_VASI • NDD_2_HEA • NDD_2_IPV6_LSO • NDD_2_IPV6_CSO • NDD_2_IPV6_PARTIAL_CSO • NDD_2_IPV4_PARTIAL_CSO • NDD_2_LARGE_RECEIVE • NDD_2_ARPINPUT • NDD_2_ECHAN_ELEM • NDD_2_SEA_ELEM, • NDD_2_ROCE • NDD_2_VIRTUAL_PORT • NDD_2_PHYS_LINK_UP • NDD_2_VNIC Эти значения доступны в виде символьных констант. **Примечание:** Поскольку заданы не все возможные значения расширения поля флага, значение может содержать другие опции.	оба варианта
vlan_id	unsigned int	Идентификатор виртуальной LAN (VLAN) (ИД VLAN указывается в разрядах 0 - 11).	оба варианта
vlan_pri	unsigned int	Приоритет VLAN (приоритет VLAN указывается в разрядах 13 - 15).	оба варианта
alias	String[16]	Имя псевдонима сетевого адаптера.	Асинхронно
nobufs	unsigned long long	Число неудачных запросов на получение сетевых буферов (MBUF) от драйвера устройства.	Асинхронно
tx_packets	unsigned long long	Число пакетов, успешно переданных сетевым устройством.	Асинхронно
tx_bytes	unsigned long long	Число байт, успешно переданных сетевым устройством.	Асинхронно
tx_interrupts	unsigned long long	Число прерываний передачи, полученных драйвером от адаптера.	Асинхронно
tx_errors	unsigned long long	Число ошибок передачи сетевого устройства. Это число учитывает аппаратные ошибки и ошибки в работе сети.	Асинхронно
tx_packets_dropped	unsigned long long	Число отброшенных пакетов во время передачи данных. Число пакетов, принятых драйвером устройства для отправки, но не переданных устройству.	Асинхронно
tx_queue_overflow	unsigned long long	Число переполнений программной очереди передачи.	Асинхронно
tx_queue_size	unsigned long long	Максимальное число пакетов, находившихся в программной очереди передачи.	Асинхронно
tx_queue_len	unsigned long long	Число ожидающих исходящих пакетов в текущих программных и аппаратных очередях передачи.	Асинхронно
tx_broadcast_packets	unsigned long long	Число переданных пакетов широковещательной рассылки.	Асинхронно
tx_multicast_packets	unsigned long long	Число переданных пакетов многоцелевой рассылки.	Асинхронно
tx_carrier_sense	unsigned long long	Число неудачных передач, связанных с отсутствием контроля несущей.	Асинхронно
tx_DMA_underrun	unsigned long long	Число неудачных передач, связанных с опустошением DMA (Прямой доступ к памяти).	Асинхронно
tx_lost_CTS_errors	unsigned long long	Число неудачных передач, связанных с потерей сигнала готовности к приему.	Асинхронно
tx_timeout_errors	unsigned long long	Число неудачных передач, связанных с ошибками тайм-аута на уровне сетевого адаптера.	Асинхронно

Таблица 8. Статистика драйвера сетевого устройства (продолжение)

Имя поля сценария Vue	Тип данных	Описание	Тип доступа к полю (Асинхронный или Оба)
tx_max_collision_errors	unsigned long long	Число неудачных передач, связанных с конфликтами переданных пакетов. В этом случае число обнаруженных конфликтов переданных пакетов превысило число повторных передач на уровне сетевого адаптера.	Асинхронно
tx_late_collision_errors	unsigned long long	Число неудачных передач, связанных с конфликтом после начала передачи.	Асинхронно
tx_deferred	unsigned long long	Число пакетов, передача которых была отложена.	Асинхронно
tx_hw_q_len	unsigned long long	Текущее число исходящих пакетов в аппаратной очереди передачи.	Асинхронно
tx_sw_q_len	unsigned long long	Текущее число исходящих пакетов в программной очереди передачи.	Асинхронно
tx_single_collision_count	unsigned long long	Число отдельных конфликтов в ходе передачи.	Асинхронно
tx_multiple_collision_count	unsigned long long	Число множественных конфликтов в ходе передачи.	Асинхронно
sqe_test	unsigned long long	Число успешно выполненных во время передачи тестов качества сигнала (SQE).	Асинхронно
ucast_pkts_reqs	unsigned long long	Число исходящих пакетов одноадресной рассылки, запрошенных сетевым устройством.	Асинхронно
mcast_pkts_reqs	unsigned long long	Число исходящих пакетов многоадресной рассылки, запрошенных сетевым устройством.	Асинхронно
bcast_pkts_reqs	unsigned long long	Число исходящих пакетов широковещательной рассылки, запрошенных сетевым устройством.	Асинхронно
rx_packets	unsigned long long	Число пакетов, успешно принятых сетевым устройством.	Асинхронно
rx_bytes	unsigned long long	Число байт, успешно принятых сетевым устройством.	Асинхронно
rx_interrupts	unsigned long long	Число прерываний приема, полученных драйвером от адаптера.	Асинхронно
rx_errors	unsigned long long	Число ошибок, возникших при приеме данных через данное устройство. Это число учитывает аппаратные ошибки и ошибки в работе сети.	Асинхронно
rx_packets_dropped	unsigned long long	Число отброшенных пакетов во время получения данных. Число пакетов, полученных драйвером устройства от этого устройства, которые не были переданы сетевому демультиплексору.	Асинхронно
rx_bad_packets	unsigned long long	Число неправильных пакетов, полученных драйвером устройства.	Асинхронно
rx_broadcast_packets	unsigned long long	Число принятых пакетов широковещательной рассылки.	Асинхронно
rx_multicast_packets	unsigned long long	Число полученных пакетов многоадресной рассылки.	Асинхронно
rx_noresource_errors	unsigned long long	Число полученных пакетов, отброшенных на аппаратном уровне из-за ошибки, связанной с отсутствием ресурса.	Асинхронно
rx_alignment_errors	unsigned long long	Число входящих пакетов с ошибками выравнивания.	Асинхронно
rx_DMA_overrun	unsigned long long	Число входящих пакетов с ошибками переполнения DMA.	Асинхронно
rx_CRC_errors	unsigned long long	Число входящих пакетов с ошибками контрольной суммы.	Асинхронно
rstart_cnt	unsigned long long	Сколько раз было выполнено восстановление после ошибки адаптера.	Асинхронно
rx_collision_errors	unsigned long long	Число входящих пакетов, при приеме которых произошел конфликт.	Асинхронно
rx_packet_tooshort_errors	unsigned long long	Число входящих пакетов с ошибкой длины (размер пакета меньше минимального размера пакета Ethernet).	Асинхронно
rx_packet_toolong_errors	unsigned long long	Число входящих пакетов с ошибкой длины (размер пакета превышает максимальный размер пакета Ethernet).	Асинхронно
rx_packets_discardedbyadapter	unsigned long long	Число полученных пакетов, отброшенных адаптеров по другим причинам.	Асинхронно
rx_start	unsigned long long	Число запусков получателя на уровне адаптера.	Асинхронно

Статистика на основе сетевого интерфейса

В следующей таблице перечислены имена поддерживаемых полей для статистики сетевого интерфейса. Эти поля относятся к экземплярам устройств сетевых интерфейсов. Статистику сетевого интерфейса можно получить как в синхронном, так и в асинхронном режиме. К статистике сетевого интерфейса можно обратиться с помощью `"__stat.net.interface.<en0...n>->имя-поля`

Таблица 9. Статистика на основе сетевого интерфейса

Имя поля сценария Vue	Тип данных	Описание
mtu	unsigned long long	Максимальный блок передачи. Максимальный размер передаваемого через интерфейс пакета (в байтах).
flags	unsigned long long	Флаг интерфейса. Возможные значения этого флага: • IFF_UP • IFF_BROADCAST • IFF_DEBUG • IFF_LOOPBACK • IFF_POINTOPOINT • IFF_VIPA • IFF_NOTRAILERS • IFF_RUNNING • IFF_PROMISC • IFF_NOARP Эти значения доступны в виде символьных констант. **Примечание:** Поскольку заданы не все возможные значения флагов интерфейса, значение может содержать другие опции.
type	unsigned int	Тип интерфейса. Возможные значения этого поля: • IFT_ETHER • IFT_IB • IFT_LOOP • IFT_FDDI • IFT_ISO88023 • IFT_ATM • IFT_OTHER • IFT_TUNNEL Эти значения доступны в виде символьных констант. **Примечание:** Поскольку заданы не все возможные значения флагов интерфейса, значение может содержать другие опции.
ipackets	unsigned long long	Число пакетов, полученных этим сетевым интерфейсом.
ibytes	unsigned long long	Число байт, принятых этим сетевым интерфейсом.
ierrors	unsigned long long	Число ошибок ввода. Это число учитывает пакеты неверного формата, неправильные контрольные суммы, а также случаи переполнения буфера драйвера устройства.
opackets	unsigned long long	Число пакетов, переданных этим сетевым интерфейсом.
obytes	unsigned long long	Число байт, переданных этим сетевым интерфейсом.
oerrors	unsigned long long	Число ошибок вывода. Это значение учитывает сбои при установлении соединения и случаи переполнения очереди вывода адаптера.
collisions	unsigned long long	Число конфликтов пакетов, обнаруженных в интерфейсах CSMA (Множественный доступ с контролем несущей).
if_arpdrops	unsigned long long	Отброшено, поскольку нет ответа протокола преобразования адресов (ARP).
if_iqdrops	unsigned long long	Число отброшенных пакетов во время получения данных через этот сетевой интерфейс.
index	unsigned int	Номер индекса интерфейса.

Таблица 9. Статистика на основе сетевого интерфейса (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_mcasts	unsigned long long	Число пакетов многоцелевой рассылки, переданных этим сетевым интерфейсом.
rx_mcasts	unsigned long long	Число пакетов многоцелевой рассылки, принятых этим сетевым интерфейсом.
no_proto	unsigned long long	Неподдерживаемый протокол.
bitrate	unsigned int	Скорость передачи данных по линии связи.
dev_num	unsigned long long	Номер устройства.
options	unsigned int	Поле опций. Возможные значения этого поля: • IFO_FLUSH • IFO_HIGHFUNC_QOS • IFO_MEDFUNC_QOS • IFO_MINFUNC_QOS • IFO_QOS • IFO_THREAD • IFO_LARGESEND • IFO_PKTCHAIN • IFO_AACCT • IFO_MONITOR • IFO_VIRTUAL_ETHERNET • IFO_CSO_IPV6 • IFO_LSO_IPV6 • IFO_PARTIAL_CSO_IPV6 • IFO_PARTIAL_CSO_IPV4 • IFO_RNIC • IFO_FIRSTALIAS • IFO_PSEUDO_CLUSTER Эти значения доступны в виде символьных констант. **Примечание:** Поскольку заданы не все возможные значения опций интерфейса, значение может содержать другие опции.

Статистика на основе сетевого протокола

Статистика на основе сетевого протокола

В следующей таблице перечислены имена поддерживаемых полей для статистики сетевого протокола. Эти поля относятся к указанному сетевому протоколу. К ним можно обратиться с помощью `"__stat.net.protocol.<имя-протокола>->имя-поля`. Например, к статистике для протокола IPv4 можно обратиться как `"__stat.net.protocol.ip->имя-поля"`. Эту статистику можно получить как в синхронном, так и в асинхронном режиме.

К статистике протокола IPv4 можно обратиться с помощью `"__stat.net.protocol.ip->имя-поля`

Таблица 10. Статистика на основе сетевого протокола (IPv4)

Имя поля сценария Vue	Тип данных	Описание
ipackets	unsigned long long	Полное число полученных пакетов IP.
rx_bytes	unsigned long long	Общее число байт, полученных в дейтаграммах IP.
tx_bytes	unsigned long long	Общее число байт, переданных в пакетах IP.
bad_cksum	unsigned long long	Число пакетов IP с ошибками контрольной суммы заголовка.

Таблица 10. Статистика на основе сетевого протокола (IPv4) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
shorts_pkts	unsigned long long	Размер буфера, содержащего пакет IP, меньше размера, указанного в поле общей длины в заголовке IP (общая длина включает длину заголовка IP и данных).
small_pkts	unsigned long long	Размер буфера, содержащего пакет IP, меньше размера, указанного в поле длины в заголовке IPv4.
bad_hdr_len	unsigned long long	Число пакетов IP с неправильными заголовками IP. Поле длины заголовка пакета IP содержит неправильное значение (длина заголовка IP меньше минимального размера пакета IP).
bad_data_len	unsigned long long	Число пакетов IP неправильной длины. Поле общей длины пакета IP меньше длины заголовка IP (общая длина включает длину заголовка IP и размер данных) или размер данных IP превышает максимально поддерживаемый размер пакета (IP_MAXPACKET).
bad_opts	unsigned long long	Число пакетов IP с неправильными опциями.
bad_vers	unsigned long long	Число пакетов IP с неправильным номером версии.
rx_frags	unsigned long long	Число полученных фрагментов IP.
frag_drops	unsigned long long	Число отброшенных фрагментов IP (повторяющиеся или недостаточно памяти).
frag_timeout	unsigned long long	Число фрагментов IP, отброшенных после тайм-аута.
reassembled	unsigned long long	Общее число повторно собранных пакетов IP.
forward	unsigned long long	Число пересланных пакетов IP.
no_proto	unsigned long long	Число неизвестных или неподдерживаемых пакетов.
cant_fwd	unsigned long long	Число пакетов, которые не удалось перенаправить. Пакеты получены от недоступного адресата.
tx_redirect	unsigned long long	Число переданных перенаправлений.
tx_drops	unsigned long long	Число исходящих пакетов, отброшенных из-за недоступных сетевых буферов (MBUF).
no_route	unsigned long long	Число исходящих пакетов, отброшенных из-за отсутствия маршрута.
tx_frags	unsigned long long	Число созданных фрагментов вывода.
cant_frag	unsigned long long	Число дейтаграмм, которые не удалось фрагментировать. Задан флаг, **запрещающий фрагментацию**.
фрагментированная	unsigned long long	Число успешно фрагментированных исходящих дейтаграмм.
threads_pkts	unsigned long long	Число пакетов IP, обработанных нитями ядра (dog).
thread_drops	unsigned long long	Число пакетов IP, отброшенных нитями ядра из-за переполнения очередей.
iqueueoverflow	unsigned long long	Число пакетов IP, отброшенных из-за переполнения буфера приема сокета.
pmtu_disc	unsigned long long	Число успешных циклов вычисления маршрута MTU маршрута.
pmtu_redisc	unsigned long long	Число циклов повторного вычисления MTU маршрута.
pmtu_guesses	unsigned long long	Число попыток вычисления MTU маршрута из-за отсутствия ответа.
pmtu_timeouts	unsigned long long	Число тайм-аутов ответов вычисления MTU маршрута.
pmtu_decs	unsigned long long	Число обнаруженных уменьшений вычисления MTU маршрута.
tx_pmtu_pkts	unsigned long long	Число пакетов вычисления MTU маршрута.
pmtu_nomem	unsigned long long	Число сбоев выделения памяти для вычисления MTU маршрута.

Таблица 10. Статистика на основе сетевого протокола (IPv4) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_dgd_pkts	unsigned long long	Число переданных пакетов функции обнаружения сбоев в работе шлюза (DGD).
dgd_nomem	unsigned long long	Число пакетов функции обнаружения сбоев в работе шлюза (DGD), которые не были переданы вследствие ошибок выделения.
dgd_nogw	unsigned long long	Число шлюзов обнаружения сбоев в работе шлюза (DGD), которые не были добавлены вследствие ошибок выделения.
bad_src	unsigned long long	Число пакетов с недопустимым исходным адресом.
delivered	unsigned long long	Число загруженных пакетов IP.
tx_local	unsigned long long	Полное число созданных пакетов IP.
tx_raw	unsigned long long	Полное число созданных необработанных пакетов IP.
hdr_errs	unsigned long long	Число ошибок заголовков.
addr_errs	unsigned long long	Число дейтаграмм с ошибками IP-адресов.
rx_discards	unsigned long long	Число отброшенных входящих дейтаграмм.
mcast_addr_errs	unsigned long long	Число пакетов многоцелевой рассылки IP, отброшенных из-за отсутствия получателя.
rx_mcast_bytes	unsigned long long	Количество полученных байт многоцелевой рассылки IP.
tx_mcast_bytes	unsigned long long	Количество переданных байт многоцелевой рассылки IP.
rx_mcast_pkts	unsigned long long	Число полученных дейтаграмм многоцелевой рассылки IP.
tx_mcast_pkts	unsigned long long	Число переданных дейтаграмм многоцелевой рассылки IP.
rx_bcast_pkts	unsigned long long	Число полученных дейтаграмм широковещательной рассылки IP.
tx_bcast_pkts	unsigned long long	Число переданных дейтаграмм широковещательной рассылки IP.
tx_mls_drops	unsigned long long	Число исходящих пакетов IP, отброшенных фильтрами многоуровневой защиты (MLS).
rx_mls_drops	unsigned long long	Число входящих пакетов IP, отброшенных фильтрами MLS.

К статистике протокола IPv6 можно обратиться с помощью `"__stat.net.protocol.ipv6->имя-поля`

Таблица 11. Статистика на основе сетевого протокола (IPv6)

Имя поля сценария Vue	Тип данных	Описание
ipackets	unsigned long long	Полное число полученных пакетов IPv6.
rx_bytes	unsigned long long	Общее число байт, полученных в дейтаграммах IPv6.
tx_bytes	unsigned long long	Общее число байт, переданных в пакетах IPv6.
raw_cksum	unsigned long long	Число пакетов IPv6, которые не были доставлены из-за ошибок в контрольной сумме IPv6.
shorts_pkts	unsigned long long	В буфере MBUF недостаточно места для размещения пакета IPv6 (заголовок IPv6 и данные).
small_pkts	unsigned long long	В буфере MBUF недостаточно места для размещения заголовка IPv6.
rx_nomem	unsigned long long	Сколько раз сетевые буферы (MBUF) были недоступны для входящих пакетов.
tx_nomen	unsigned long long	Сколько раз сетевые буферы (MBUF) были недоступны для исходящих пакетов.
no_proto	unsigned long long	Число неизвестных или неподдерживаемых пакетов.

Таблица 11. Статистика на основе сетевого протокола (IPv6) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
bad_vers	unsigned long long	Число пакетов IPv6 с неправильным номером версии.
rx_frags	unsigned long long	Число полученных фрагментов IPv6.
frag_drops	unsigned long long	Число отброшенных фрагментов IPv6 (повторяющиеся или недостаточно памяти).
frag_timeout	unsigned long long	Число фрагментов IPv6, отброшенных после тайм-аута.
фрагментированная	unsigned long long	Число успешно фрагментированных исходящих дейтаграмм.
tx_frags	unsigned long long	Число созданных фрагментов вывода.
reassembled	unsigned long long	Общее число повторно собранных пакетов IPv6.
cant_frag	unsigned long long	Число дейтаграмм, которые не удалось фрагментировать. Задан флаг, **запрещающий фрагментацию**.
forward	unsigned long long	Число пересланных пакетов IPv6.
cant_fwd	unsigned long long	Число пакетов, которые не удалось перенаправить. Пакеты получены от недоступного адресата.
bad_src	unsigned long long	Число пакетов с недопустимым исходным адресом.
tx_drops	unsigned long long	Число исходящих пакетов, отброшенных из-за недоступных сетевых буферов (MBUF).
no_route	unsigned long long	Число исходящих пакетов, отброшенных из-за отсутствия маршрута.
delivered	unsigned long long	Число загруженных пакетов IPv6.
tx_local	unsigned long long	Полное число созданных пакетов IPv6.
iqueueoverflow	unsigned long long	Число пакетов IPv6, отброшенных из-за переполнения буфера приема сокета.
big_pkts	unsigned long long	Число пакетов IPv6, которые не были перенаправлены из-за превышения MTU.
tx_raw	unsigned long long	Общее число созданных необработанных пакетов IPv6.
hdr_errs	unsigned long long	Число ошибок заголовков.
addr_errs	unsigned long long	Число дейтаграмм с ошибками адресов IPv6.
rx_discards	unsigned long long	Число отброшенных входящих дейтаграмм.
rx_mcast_bytes	unsigned long long	Количество полученных байт многоцелевой рассылки IPv6.
tx_mcast_bytes	unsigned long long	Количество переданных байт многоцелевой рассылки IPv6.
rx_mcast_pkts	unsigned long long	Число полученных дейтаграмм многоцелевой рассылки IPv6.
tx_mcast_pkts	unsigned long long	Число переданных дейтаграмм многоцелевой рассылки IPv6.
rx_bcast_pkts	unsigned long long	Число полученных дейтаграмм широковещательной рассылки IPv6.
tx_bcast_pkts	unsigned long long	Число переданных дейтаграмм широковещательной рассылки IPv6.

Таблица 11. Статистика на основе сетевого протокола (IPv6) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_mls_drops	unsigned long long	Число исходящих пакетов IPv6, отброшенных фильтрами MLS.
rx_mls_drops	unsigned long long	Число входящих пакетов IPv6, отброшенных фильтрами MLS.

К статистике протокола TCP можно обратиться с помощью "`__stat.net.protocol.tcp->имя-поля`

Таблица 12. Статистика на основе сетевого протокола (TCP)

Имя поля сценария Vue	Тип данных	Описание
tx_total	unsigned long long	Общее число переданных пакетов TCP. Это значение учитывает пакеты **data** и **ack**.
rx_total	unsigned long long	Полное число полученных пакетов TCP. Это значение учитывает пакеты **data** и **ack**.
opackets	unsigned long long	Число переданных пакетов данных TCP.
ipackets	unsigned long long	Число полученных пакетов данных TCP.
tx_bytes	unsigned long long	Число переданных байт данных TCP.
rx_bytes	unsigned long long	Число последовательно полученных байт данных TCP.
rexmit_pkts	unsigned long long	Число повторно переданных пакетов данных TCP.
rexmit_bytes	unsigned long long	Число повторно переданных байт данных TCP.
tx_ack_pkts	unsigned long long	Число переданных пакетов ACK TCP.
rx_ack_pkts	unsigned long long	Число полученных пакетов ACK TCP.
rx_ack_bytes	unsigned long long	Число полученных байт ACK TCP.
rx_dup_pkts	unsigned long long	Число полученных полностью повторяющихся пакетов TCP.
rx_dup_bytes	unsigned long long	Число полученных полностью повторяющихся байт TCP.
rx_part_dup_pkts	unsigned long long	Число полученных пакетов с частично повторяющимися данными.
rx_part_dup_bytes	unsigned long long	Число повторяющихся байт, полученных из частично совпадающих пакетов.
rx_dup_ack_pkts	unsigned long long	Число полученных повторяющихся пакетов ACK TCP.
tx_win_probe	unsigned long long	Число переданных пакетов тестов окна TCP.
rx_win_probe	unsigned long long	Число полученных пакетов тестов окна TCP.
tx_win_update	unsigned long long	Число переданных пакетов обновления окна TCP.
rx_win_update	unsigned long long	Число полученных пакетов обновления окна TCP.
tx_delay_ack_pkts	unsigned long long	Число переданных с задержкой пакетов ACK TCP.
tx_urg_pkts	unsigned long long	Число переданных пакетов URG.
tx_ctrl_pkts	unsigned long long	Число переданных пакетов управления (SYN|FIN|RST).
tx_large_send_pkts	unsigned long long	Число переданных больших пакетов.
tx_large_send_bytes	unsigned long long	Число байт, переданных с помощью опции отправки больших пакетов.

Таблица 12. Статистика на основе сетевого протокола (TCP) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_large_send_max	unsigned long long	Максимальное число байт, которые можно передать с помощью опции отправки больших пакетов.
rx_ack_unsent_data	unsigned long long	Число ACK, полученных для неотправленных данных.
rx_out_order_pkts	unsigned long long	Число полученных вне очереди пакетов.
rx_out_order_bytes	unsigned long long	Число полученных вне очереди байт.
rx_after_close_pkts	unsigned long long	Число пакетов, полученных после закрытия соединения.
fast_lo_conns	unsigned long long	Число циклических соединений быстрого доступа.
tx_fast_lo_pkts	unsigned long long	Число пакетов, переданных через циклические соединения быстрого доступа.
rx_fast_lo_pkts	unsigned long long	Число пакетов, полученных через циклические соединения быстрого доступа.
tx_fast_lo_bytes	unsigned long long	Число байт, переданных через циклические соединения быстрого доступа.
rx_fast_lo_bytes	unsigned long long	Число байт, полученных через циклические соединения быстрого доступа.
rx_bad_hw_cksum	unsigned long long	Число полученных пакетов с ошибками аппаратной контрольной суммы.
rx_bad_cksum	unsigned long long	Число пакетов, отброшенных вследствие ошибок контрольной суммы.
rx_bad_off	unsigned long long	Число пакетов, отброшенных вследствие неправильных полей смещения ошибок.
rx_short_pkts	unsigned long long	Число отброшенных слишком коротких пакетов. Размер пакета меньше минимального размера пакета TCP.
rx_queue_ovflow	unsigned long long	Число пакетов, отброшенных из-за переполнения очереди получателя.
rx_after_win_pkts	unsigned long long	Число полученных пакетов, данные которых превышают размер окна получателя.
rx_after_win_bytes	unsigned long long	Число полученных байт, данные которых превышают размер окна получателя.
initiated	unsigned long long	Число запросов для соединения TCP.
accepted	unsigned long long	Число принятых соединений TCP.
established	unsigned long long	Число установленных соединений TCP.
closed	unsigned long long	Число закрытых соединений TCP, включая отброшенные соединения.
dropped	unsigned long long	Число отброшенных соединений TCP.
ecn_conns	unsigned long long	Число соединений с поддержкой функции явного уведомления о перегрузке (ECN).
ecn_congestion	unsigned long long	Число ответов ECN.
conn_drops	unsigned long long	Число отброшенных неустановленных соединений.
segs_timed	unsigned long long	Число попыток сегментов обновить время оборота пакета (RTT).
rtt_updated	unsigned long long	Сколько раз сегменты обновили RTT.
ecnce	unsigned long long	Число сегментов с флагом перегрузки (CE).
ecnwr	unsigned long long	Число сегментов с флагом уменьшенного окна перегрузки (CWR).

Таблица 12. Статистика на основе сетевого протокола (TCP) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
pmtu_resends	unsigned long long	Число повторных передач, связанных с вычислением MTU маршрута.
pmtu_halts	unsigned long long	Число прерванных операций вычисления MTU маршрута из-за повторной передачи.
rexmt_timeout	unsigned long long	Число тайм-аутов повторной передачи.
timeout_drops	unsigned long long	Число соединений, отброшенных из-за тайм-аутов повторной передачи.
fast_rxmt	unsigned long long	Число быстрых операций повторной передачи.
new_reno_rxmt	unsigned long long	Число быстрых операций повторной передачи NewReno.
false_fast_rxmt	unsigned long long	Число предотвращенных ложных быстрых повторных передач.
persist_timeouts	unsigned long long	Число тайм-аутов сохранения.
persist_drops	unsigned long long	Число соединений, отброшенных из-за тайм-аутов сохранения.
keep_alive_timeout	unsigned long long	Число тайм-аутов проверки активности.
keep_alive_probe	unsigned long long	Число переданных контрольных тестов.
keep_alive_drops	unsigned long long	Число соединений, отброшенных функцией проверки активности.
delay_ack_syn	unsigned long long	Число отложенных ACK для SYN.
delay_ack_fin	unsigned long long	Число отложенных ACK для FIN.
sack_blocks_upd	unsigned long long	Число расширений массива блоков выборочных подтверждений (SACK).
sack_holes_upd	unsigned long long	Число расширений массива дыр SACK.
tx_drops	unsigned long long	Число пакетов, отброшенных из-за ошибок выделения памяти.
time_wait_reuse	unsigned long long	Число повторных использований существующего соединения в состоянии TIME_WAIT для нового исходящего соединения.
send_and_disc	unsigned long long	Число отправкой и отключений.
spliced_conns	unsigned long long	Число сложных соединений TCP.
splice_closed	unsigned long long	Число закрытых сложных соединений TCP.
splice_resets	unsigned long long	Число сброшенных сложных соединений TCP.
splice_timeouts	unsigned long long	Число тайм-аутов сложных соединений TCP.
splice_persist_drops	unsigned long long	Число постоянных тайм-аутов сложных соединений TCP.
splice_keep_drops	unsigned long long	Число тайм-аутов активности сложных соединений TCP.
bad_ack_conn_drops	unsigned long long	Число соединений, отброшенных из-за неправильных ACK.
dup_syn_conn_drops	unsigned long long	Число соединений, отброшенных из-за совпадающих пакетов SYN.
auto_cksum_offload	unsigned long long	Число соединений, для которых аппаратная проверка контрольной суммы была выключена в динамическом режиме.
bad_syn	unsigned long long	Число недопустимых пакетов, отброшенных получателями.

Таблица 12. Статистика на основе сетевого протокола (TCP) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
limit_transmit	unsigned long long	Число быстрых операций повторной передачи, выполненных с помощью алгоритма **ограниченной передачи**.
pred_acks	unsigned long long	Число правильных предсказаний заголовков пакетов ACK.
pred_dat	unsigned long long	Число правильных предсказаний заголовков пакетов данных.
paws_drops	unsigned long long	Число сегментов, отброшенных из-за PAWS.
persist_drops	unsigned long long	Число отброшенных соединений в постоянном состоянии.
fake_syn_drops	unsigned long long	Число отброшенных фиктивных сегментов SYN.
fake_rst_drops	unsigned long long	Число отброшенных фиктивных сегментов RST.
data_inject_drops	unsigned long long	Число отброшенных сегментов внедрения данных.
tr_max_conn_drops	unsigned long long	Максимальное число соединений, отброшенных для управления потоком данных TCP.
tr_nomem_drops	unsigned long long	Число соединений, отброшенных процессом управления потоком данных из-за нехватки памяти.
tr_max_per_host	unsigned long long	Максимальное число соединений на хост, отброшенных для управления потоком данных.

К статистике протокола UDP можно обратиться с помощью "`__stat.net.protocol.udp->имя-поля`

Таблица 13. Статистика на основе сетевого протокола (UDP)

Имя поля сценария Vue	Тип данных	Описание
opackets	unsigned long long	Общее число переданных дейтаграмм UDP.
ipackets	unsigned long long	Общее число полученных дейтаграмм UDP.
hdr_drops	unsigned long long	Число пакетов, размер которых меньше размера заголовка. Заголовки IP и UDP нельзя разместить в одном буфере памяти (MBUF).
bad_cksum	unsigned long long	Число полученных пакетов UDP с неправильными контрольными суммами.
bad_len	unsigned long long	Число полученных пакетов неправильной длины. Длина UDP, указанная в пакете, больше общего размера пакета из заголовка IP или меньше размера заголовка UDP.
no_socket	unsigned long long	Число пакетов, отброшенных из-за отсутствия сокетов на уровне порта.
sock_buf_overflow	unsigned long long	Число переполнений буфера сокета.
dgm_no_socket	unsigned long long	Число дейтаграмм широковещательной или многоцелевой рассылки, отброшенных из-за отсутствия сокета.
pcb_cache_miss	unsigned long long	Число входящих пакетов, отсутствующих в кэше PCB.

К статистике протокола управляющих сообщений Internet (ICMP) можно обратиться с помощью "__stat.net.protocol.icmp->имя-поля

Таблица 14. Статистика на основе сетевого протокола (ICMP)

Имя поля сценария Vue	Тип данных	Описание
sent	unsigned long long	Общее число переданных пакетов ICMP.
received	unsigned long long	Полное число полученных пакетов ICMP.
errors	unsigned long long	Число ошибок ICMP.
bad_cksum	unsigned long long	Число полученных сообщений ICMP с неправильными контрольными суммами.
bad_len	unsigned long long	Число полученных сообщений ICMP неправильной длины.
bad_code	unsigned long long	Число полученных сообщений ICMP с неправильными кодовыми полями. В этих сообщениях **icmp_code** лежит за пределами допустимого диапазона.
old_msg	unsigned long long	Число ошибок, которые не были созданы из-за протокола ICMP старого пакета.
old_short_msg	unsigned long long	Число ошибок, которые не были созданы из-за слишком короткого старого пакета IP.
short_msg	unsigned long long	Размер сообщения ICMP меньше минимальной длины сообщения ICMP (размер пакета <ICMP_MINLEN).
reflect	unsigned long long	Число созданных ответных сообщений ICMP.

К статистике протокола ICMPV6 можно обратиться с помощью "__stat.net.protocol.icmpv6->имя-поля

Таблица 15. Статистика на основе сетевого протокола (ICMPV6)

Имя поля сценария Vue	Тип данных	Описание
tx_echo_reply	unsigned long long	Общее число переданных эхо-ответов ICMPv6.
rx_echo_reply	`unsigned long long	Общее число полученных эхо-ответов ICMPv6.
errors	unsigned long long	Число ошибок ICMPv6.
rx_bad_cksum	unsigned long long	Число полученных сообщений ICMPv6 с неправильными контрольными суммами.
rx_bad_len	unsigned long long	Число полученных сообщений ICMPv6 неправильной длины.
bad_code	unsigned long long	Число полученных сообщений ICMPv6 с неправильными кодовыми полями. В этих сообщениях **icmp6_code** лежит за пределами допустимого диапазона.
old_msg	unsigned long long	Число ошибок, которые не были созданы из-за протокола ICMPv6 старого пакета.
short_msg	unsigned long long	Размер сообщения ICMPv6 меньше минимальной длины сообщения ICMPv6 (размер пакета <ICMP6_MINLEN).
reflect	unsigned long long	Число созданных ответных сообщений ICMPv6.
err_rate_limit	unsigned long long	Число ошибок ICMPv6, превышающих ограничение частоты ошибок.
tx_unreach	unsigned long long	Число переданных недоступных сообщений.
rx_unreach	unsigned long long	Число полученных недоступных сообщений.

Таблица 15. Статистика на основе сетевого протокола (ICMPV6) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_big_pkt	unsigned long long	Число переданных больших пакетов ICMPv6.
rx_big_pkt	unsigned long long	Число полученных больших пакетов ICMPv6.
tx_timxceed	unsigned long long	Число превышений времени отправки сообщений ICMPv6.
rx_timxceed	unsigned long long	Число превышений времени получения сообщений ICMPv6.
tx_param_prob	unsigned long long	Число неполадок параметров при отправке сообщений ICMPv6.
rx_param_prob	unsigned long long	Число неполадок параметров при получении сообщений ICMPv6.
tx_echo_req	unsigned long long	Число отправленных эхо-запросов.
rx_echo_req	unsigned long long	Число полученных эхо-запросов.
tx_mld_qry	unsigned long long	Число отправленных групповых запросов.
rx_mld_qry	unsigned long long	Число полученных групповых запросов.
tx_mld_report	unsigned long long	Число отправленных групповых отчетов.
rx_mld_report	unsigned long long	Число полученных групповых отчетов.
rx_bad_mld_qry	unsigned long long	Число полученных неправильных групповых запросов.
rx_bad_mld_report	unsigned long long	Число полученных неправильных групповых отчетов.
rx_our_mld_report	unsigned long long	Число полученных собственных групповых отчетов.
tx_mld_term	unsigned long long	Число отправленных групповых прерываний.
rx_mld_term	unsigned long long	Число полученных групповых прерываний.
rx_bad_mld_term	unsigned long long	Число полученных неправильных групповых прерываний.
tx_redirect	unsigned long long	Число переданных перенаправлений.
rx_redirect	unsigned long long	Число полученных перенаправлений.
rx_bad_redirect	unsigned long long	Число полученных неправильных перенаправлений.
tx_router_sol	unsigned long long	Число отправленных запросов маршрутизаторов.
rx_router_sol	unsigned long long	Число полученных запросов маршрутизаторов.
rx_bad_router_sol	unsigned long long	Число полученных неправильных запросов маршрутизаторов.
tx_router_adv	unsigned long long	Число отправленных объявлений маршрутизаторов.
rx_router_adv	unsigned long long	Число полученных объявлений маршрутизаторов.
rx_bad_router_adv	unsigned long long	Число полученных неправильных объявлений маршрутизаторов.
tx_nd_sol	unsigned long long	Число отправленных запросов соседних систем.
rx_nd_sol	unsigned long long	Число полученных запросов соседних систем.
rx_bad_nd_sol	unsigned long long	Число полученных неправильных запросов соседних систем.

Таблица 15. Статистика на основе сетевого протокола (ICMPV6) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
tx_nd_adv	unsigned long long	Число отправленных объявлений соседних систем.
rx_nd_adv	unsigned long long	Число полученных объявлений соседних систем.
rx_bad_nd_adv	unsigned long long	Число полученных неправильных объявлений соседних систем.
tx_router_renum	unsigned long long	Число отправленных перенумераций маршрутизаторов.
rx_router_renum	unsigned long long	Число полученных перенумераций маршрутизаторов.
tx_haad_req	unsigned long long	Число отправленных запросов HAAD.
rx_haad_req	unsigned long long	Число полученных запросов HAAD.
rx_bad_haad_req	unsigned long long	Число полученных неправильных запросов HAAD.
tx_haad_reply	unsigned long long	Число отправленных ответов HAAD.
rx_haad_reply	unsigned long long	Число полученных ответов HAAD.
rx_bad_haad_reply	unsigned long long	Число полученных неправильных ответов HAAD.
tx_prefix_sol	unsigned long long	Число отправленных запросов префикса.
rx_prefix_sol	unsigned long long	Число полученных запросов префикса.
rx_bad_prefix_sol	unsigned long long	Число полученных неправильных запросов префикса.
tx_prefix_adv	unsigned long long	Число отправленных объявлений префикса.
rx_prefix_adv	unsigned long long	Число полученных объявлений префикса.
rx_bad_prefix_adv	unsigned long long	Число полученных неправильных объявлений префикса.
no_mobility	unsigned long long	Число вызовов переадресации до запуска.
ndp_q_drops	unsigned long long	Число блокированных пакетов, отброшенных при ожидании завершения ndp.

К статистике протокола управления группами Интернета (IGMP) можно обратиться с помощью `"__stat.net.protocol.igmp->имя-поля`

Таблица 16. Статистика на основе сетевого протокола (IGMP)

Имя поля сценария Vue	Тип данных	Описание
rx_total	unsigned int	Общее число полученных сообщений IGMP.
rx_queries	unsigned int	Число полученных запросов о членстве IGMP.
tx_reports	unsigned int	Число переданных отчетов о членстве IGMP.
rx_reports	unsigned int	Число полученных отчетов о членстве IGMP.
rx_our_reports	unsigned int	Число полученных отчетов о членстве IGMP для собственных групп.
rx_bad_cksum	unsigned int	Число полученных сообщений IGMP с неправильными контрольными суммами.
rx_short_msg	unsigned int	Число полученных сообщений IGMP с данными, размер которых меньше минимального размера сообщения IGMP.

Таблица 16. Статистика на основе сетевого протокола (IGMP) (продолжение)

Имя поля сценария Vue	Тип данных	Описание
rx_bad_queries	unsigned int	Число полученных запросов о членстве IGMP с недопустимыми полями.
rx_bad_reports	unsigned int	Число полученных отчетов о членстве IGMP с недопустимыми полями.

К статистике протокола преобразования адресов (ARP) можно обратиться с помощью `"__stat.net.protocol.arp->имя-поля`

Таблица 17. Статистика на основе сетевого протокола (ARP)

Имя поля сценария Vue	Тип данных	Описание
purged	unsigned int	Число очищенных пакетов ARP. Если в пакете нет места, то из него удаляются самые старые записи ARP.
sent	unsigned int	Общее число переданных пакетов ARP.

Статистика памяти

В отличие от прочих показателей статистика памяти столь же точна, как и показатели, полученные в синхронном режиме, но при сборе статистики не накапливаются погрешности, связанные с повторным получением данных во время выполнения сценария.

Некоторые показатели памяти, такие как число страничных ошибок, меняются очень часто. Поэтому сбор статистики памяти в асинхронном режиме приводит к неверным данным. Например, для сбора статистики памяти для процесса, в котором происходит ошибка из-за нехватки памяти, можно использовать интервал между системными вызовами `fork` и `exit` для сбора данных. Если показатели памяти не были собраны в том же самом интервале, то статистика памяти остается неизменной.

Доступ к показателям памяти возможен в следующем формате:

`__stat.mem->`*<поле>*

Например, для доступа к числу страничных ошибок в течение заданного интервала выберите:

```
__stat.mem->page_faults
```

Здесь `__stat` представляет показатель, а `mem` указывает, что этот показатель относится к памяти.

Для показателей предусмотрены следующие поля:

Таблица 18. Поля ProbeVue для показателей памяти

Имя поля сценария Vue	Тип данных	Описание
page_faults	unsigned long long	Число страничных ошибок.
page_reclaims	unsigned long long	Число рекламаций страниц.
lock_misses	unsigned long long	Число неудачных блокировок.
back_tracks	unsigned long long	Число операций отступления.
pageins	unsigned long long	Число страниц, загруженных в операциях подкачки, во время сбора статистики памяти.
pageouts	unsigned long long	Число страниц, выгруженных в операциях подкачки, во время сбора статистики памяти.
num_ios	unsigned long long	Число операций начала ввода-вывода.
num_iodone	unsigned long long	Число операций `iodone`.
zerofills	unsigned long long	Число страниц, заполненных нулем.

Таблица 18. Поля ProbeVue для показателей памяти (продолжение)

Имя поля сценария Vue	Тип данных	Описание
exec_fills	unsigned long long	Число страниц, заполненных исполняемыми файлами.
page_scans	unsigned long long	Число просматриваемых страниц.
pager_cycles	unsigned long long	Число циклов стрелки часов пейджера.
page_steals	unsigned long long	Число наименее используемых страниц, которые должны быть включены в список свободных страниц.
free_frame_waits	unsigned long long	Число страниц оперативной памяти, которые должны быть добавлены в список доступных страниц памяти.
extnd_xpt_waits	unsigned long long	Число операций extnd_xpt_wait.
pending_io_waits	unsigned long long	Число незавершенных операций ожидания ввода-вывода.

Статистика CPU

Вся статистика для CPU сгруппирована в новый уровень, называющийся *cpu*. Все показатели статистики CPU начинаются с `__stat.cpu`. За этим элементом следует или экземпляр CPU, например, cpu0, cpu1 и т. д., или подуровень `cpu_total`.

Статистика по логическим процессорам:

Статистика по логическим процессорам собирается для каждого отдельного логического процессора. В сценарии Vue каждый показатель указывается в следующем формате:

`__stat.cpu.cpuX->`*`имя-поля`*

Здесь `cpuX` представляет экземпляр процессора, например, cpu0, cpu1, а *имя-поля* - это идентификатор показателя. Эти поля аналогичны полям структуры `perfstat_cpu_total_t`, которая определена в заголовочном файле `/usr/include/libperfstat.h`. Этот заголовочный файл - часть библиотеки AIX `perfstat`.

Таблица 19. Поддерживаемые поля ProbeVue для статистики по логическим процессорам

Имя поля сценария Vue	Тип данных	Описание
user	unsigned long long	Число тактов системных часов в пользовательском режиме.
sys	unsigned long long	Число тактов системных часов в системном режиме.
idle	unsigned long long	Число тактов системных часов в режиме бездействия.
wait	unsigned long long	Число тактов системных часов в режиме ожидания операций ввода-вывода.
pswitch	unsigned long long	Число переключений контекста.
syscall	unsigned long long	Число системных вызовов.
sysread	unsigned long long	Число системных вызовов чтения.
syswrite	unsigned long long	Число системных вызовов записи.
sysfork	unsigned long long	Число системных вызовов fork.
sysexec	unsigned long long	Число системных вызовов exec.
readch	unsigned long long	Число символов, переданных при помощи системных вызовов чтения.
writech	unsigned long long	Число символов, переданных при помощи системных вызовов записи.
bread	unsigned long long	Число прочитанных блоков.
bwrite	unsigned long long	Число записанных блоков.
lread	unsigned long long	Число логических запросов чтения.
lwrite	unsigned long long	Число логических запросов записи.

Таблица 19. Поддерживаемые поля ProbeVue для статистики по логическим процессорам (продолжение)

Имя поля сценария Vue	Тип данных	Описание
phread	unsigned long long	Число физических операций чтения, то есть операций чтения на низкоуровневых устройствах.
phwrite	unsigned long long	Число физических операций записи, то есть операций записи на низкоуровневых устройствах.
iget	unsigned long long	Число операций поиска inode.
namei	unsigned long long	Число операций поиска vnode по имени пути.
dirblk	unsigned long long	Число 512-байтовых блоков, прочитанных утилитой поиска в каталоге для нахождения записи файла.
msg	unsigned long long	Число операций для сообщений IPC.
sema	unsigned long long	Число операций для семафоров IPC.
minfaults	unsigned long long	Число страничных ошибок без ввода-вывода.
majfaults	unsigned long long	Число страничных ошибок с дисковым вводом-выводом.
puser	unsigned long long	Необработанное число тактов физического процессора в пользовательском режиме.
psys	unsigned long long	Необработанное число тактов физического процессора в системном режиме.
pidle	unsigned long long	Необработанное число тактов физического процессора в режиме простоя.
pwait	unsigned long long	Необработанное число тактов физического процессора в режиме ожидания операций ввода-вывода.
redisp_sd0	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 0 планировщика.
redisp_sd1	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 1 планировщика.
redisp_sd2	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 2 планировщика.
redisp_sd3	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 3 планировщика.
redisp_sd4	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 4 планировщика.
redisp_sd5	unsigned long long	Число повторных диспетчеризаций нитей в пределах домена сродства 5 планировщика.
migration_push	unsigned long long	Число передачи нитей из локальной очереди выполнения в другую очередь из-за исчерпания ресурсов для распределения нагрузки.
migration_S3grq	unsigned long long	Число передачи нитей из глобальной в локальную очередь выполнения в другую очередь с перемещением за пределы домена 3 планировщика.
migration_S3pul	unsigned long long	Число передачи нитей из очереди выполнения другого процессора с перемещением за пределы домена 3 планировщика.
invol_cswitch	unsigned long long	Число незапрограммированных переключений контекста нити.
vol_cswitch	unsigned long long	Число запрограммированных переключений контекста нити.
runque	unsigned long long	Число нитей в очереди выполнения.
bound	unsigned long long	Число связанных нитей.
decrintrs	unsigned long long	Число прерываний декрементного счетчика.
mpcrintrs	unsigned long long	Число прерываний *Multi-Processor Communication (MPC) receive*.
mpcsintrs	unsigned long long	Число прерываний *MPC send*.
devintrs	unsigned long long	Число прерываний устройств.
softintrs	unsigned long long	Число вызовов внеуровневых обработчиков.
phantintrs	unsigned long long	Число фантомных прерываний.
idle_donated_purr	unsigned long long	Число циклов простоя, передаваемых выделенным разделом, в котором разрешена такая передача.

Таблица 19. Поддерживаемые поля ProbeVue для статистики по логическим процессорам (продолжение)

Имя поля сценария Vue	Тип данных	Описание
idle_donated_spurr	unsigned long long	Число циклов простоя SPURR, передаваемых выделенным разделом, в котором разрешена такая передача.
busy_donated_purr	unsigned long long	Число занятых циклов, передаваемых выделенным разделом, в котором разрешена такая передача.
busy_donated_spurr	unsigned long long	Число занятых циклов SPURR, передаваемых выделенным разделом, в котором разрешена такая передача.
idle_stolen_purr	unsigned long long	Число циклов простоя, забранных гипервизором у выделенного раздела.
idle_stolen_spurr	unsigned long long	Число циклов простоя SPURR, забранных гипервизором у выделенного раздела.
busy_stolen_purr	unsigned long long	Число занятых циклов, забранных гипервизором у выделенного раздела.
busy_stolen_spurr	unsigned long long	Число занятых циклов SPURR, забранных гипервизором у выделенного раздела.
hpi	unsigned long long	Число операций подкачки в оперативную память гипервизора.
puser_spurr	unsigned long long	Число циклов SPURR в пользовательском режиме.
psys_spurr	unsigned long long	Число циклов SPURR в режиме ядра.
pidle_spurr	unsigned long long	Число циклов SPURR в режиме простоя.
pwait_spurr	unsigned long long	Число циклов SPURR в режиме ожидания.
spurrflag	int	Это поле указывает, работает ли процессор в режиме SPURR.
localdispatch	unsigned long long	Число диспетчеризаций локальных нитей в логическом процессоре.
neardispatch	unsigned long long	Число диспетчеризаций ближних нитей в логическом процессоре.
fardispatch	unsigned long long	Число диспетчеризаций дальних нитей в логическом процессоре.
cswitches	unsigned long long	Число переключений контекста.
state	int	Указывает, доступен ли процессор. Значение 0 в поле `state` означает, что процессор недоступен, отличное от нуля значение - что процессор доступен.
tb_last	unsigned long long	Последнее прочитанное значение регистра timebase.
vtb_last	unsigned long long	Последнее прочитанное значение виртуального timebase.
icount_last	unsigned long long	Последнее прочитанное значение регистра числа инструкций.

Примечание: Следующие элементы требуют доступа к памяти без закрепления: `pswitch`, `syscall`, `sysread`, `syswrite`, `sysfork`, `sysexec`, `readch`, `writech`, `bread`, `bwrite`, `lread`, `lwrite`, `phread`, `phwrite`, `iget`, `namei`, `dirblk`, `msg`, `sema`, `decrintrs`, `mpcrintrs`, `mpcsintrs`, `devintrs`, `softintrs`, `phantintrs`. Поскольку тесты сценария ProbeVue выполняются в среде, где все прерывания выключены, то в случае отсутствия необходимых данных в резидентной памяти элементам, которым требуется доступ к памяти без прикрепления, присваивается значение 0.

Общая статистика для всех процессоров: Общая статистика содержит собранную информацию о всех процессорах в системе. В сценарии Vue каждый показатель указывается в следующем формате:

`__stat.cpu.cpu_total->`*`имя-поля`*

здесь `cpu_total` - заранее определенное имя подуровня, а *имя-поля* - фактический идентификатор показателя. Эти поля аналогичны полям структуры `perfstat_cpu_total_t`, которая определена в заголовочном файле `/usr/include/libperfstat.h`. Этот заголовочный файл - часть библиотеки AIX `perfstat`.

Таблица 20. Поддерживаемые поля ProbeVue для общей статистики процессоров

Имя поля сценария Vue	Тип данных	Описание
ncpus	int	Число активных логических процессоров.
ncpus_cfg	int	Число настроенных процессоров.
description	String	Описание процессора в формате `<тип_название>`. Например, для систем с процессором POWER7 описание - POWERPC_POWER7.
processorHZ	unsigned long long	Быстродействие процессора в Гц.
user	unsigned long long	Общее число тактов системных часов в пользовательском режиме.
sys	unsigned long long	Общее число тактов системных часов в системном режиме.
простой	unsigned long long	Общее число тактов системных часов в режиме бездействия.
wait	unsigned long long	Общее число тактов системных часов в режиме ожидания операций ввода-вывода.
pswitch	unsigned long long	Число переключений контекста процесса.
syscall	unsigned long long	Число системных вызовов.
sysread	unsigned long long	Число системных вызовов чтения.
syswrite	unsigned long long	Число системных вызовов записи.
sysfork	unsigned long long	Число системных вызовов fork.
sysexec	unsigned long long	Число системных вызовов exec.
readch	unsigned long long	Число символов, переданных при помощи системных вызовов чтения.
writech	unsigned long long	Число символов, переданных при помощи системных вызовов записи.
devintrs	unsigned long long	Число прерываний устройств.
softintrs	unsigned long long	Число программных прерываний.
lbolt	unsigned long long	Число тактов системных часов, считая с последнего перезапуска операционной системы.
loadavg1	unsigned long long	Задает значение, равное произведению выражения (1<<SBITS) и среднего числа процессов, которые можно запустить, за последнюю 1 минуту. Символ << указывает, что оператор сдвига влево и значение SBITS определены в заголовочном файле `/usr/include/sys/proc.h`.
loadavg5	unsigned long long	Задает значение, равное произведению выражения (1<<SBITS) и среднего числа процессов, которые можно запустить, за последние 5 минут. Символ << указывает, что оператор сдвига влево и значение SBITS определены в заголовочном файле `/usr/include/sys/proc.h`.
loadavg15	unsigned long long	Задает значение, равное произведению выражения (1<<SBITS) и среднего числа процессов, которые можно запустить, за последние 15 минут. Символ << указывает, что оператор сдвига влево и значение SBITS определены в заголовочном файле `/usr/include/sys/proc.h`.
runque	unsigned long long	Длина очереди выполнения, то есть число процессов, готовых к запуску.
swpque	unsigned long long	Длина очереди подкачки, то есть число процессов, ожидающих подкачки в оперативную память.
bread	unsigned long long	Число прочитанных блоков.
bwrite	unsigned long long	Число записанных блоков.
lread	unsigned long long	Число логических запросов чтения.
lwrite	unsigned long long	Число логических запросов записи.
phread	unsigned long long	Число физических операций чтения, то есть операций чтения на низкоуровневых устройствах.
phwrite	unsigned long long	Число физических операций записи, то есть операций записи на низкоуровневых устройствах.
runocc	unsigned long long	Это поле обновляется при каждом обновлении очереди выполнения и указывает на то, что очередь занята. Это значение может использоваться для вычисления среднего числа готовых к запуску процессов.

Таблица 20. Поддерживаемые поля ProbeVue для общей статистики процессоров (продолжение)

Имя поля сценария Vue	Тип данных	Описание
swpocc	unsigned long long	Это поле обновляется при каждом обновлении очереди подкачки. Например, когда очередь подкачки занята. Это значение может использоваться для вычисления среднего числа ожидающих подкачки в оперативную память процессов.
iget	unsigned long long	Число операций поиска inode.
namei	unsigned long long	Число операций поиска vnode по имени пути.
dirblk	unsigned long long	Число 512-байтовых блоков, прочитанных утилитой поиска в каталоге для нахождения записи файла.
msg	unsigned long long	Число операций для сообщений IPC.
sema	unsigned long long	Число операций для семафоров IPC.
rcvint	unsigned long long	Число прерываний tty, полученных процессом.
xmtint	unsigned long long	Число прерываний передачи tty.
mdmint	unsigned long long	Число прерываний модема.
tty_rawinch	unsigned long long	Число необработанных входных символов.
tty_caninch	unsigned long long	Число канонических входных символов.
tty_rawoutch	unsigned long long	Число необработанных выходных символов.
ksched	unsigned long long	Число созданных процессов ядра.
koverf	unsigned long long	Число попыток создания процессов ядра при достигнутом пределе числа процессов, заданном в конфигурации, или при попытках пользователя создать процессы свыше установленного предела.
kexit	unsigned long long	Число процессов ядра, ставших процессами-зомби.
rbread	unsigned long long	Число запросов операций удаленного чтения.
rcread	unsigned long long	Число кэшированных операций удаленного чтения.
rbwrt	unsigned long long	Число операций удаленной записи.
rcwrt	unsigned long long	Число кэшированных операций удаленной записи.
traps	unsigned long long	Число ловушек.
ncpus_high	int	Наибольший индекс доступного процессора. Индексация процессоров начинается с 1 вместо 0.
puser	unsigned long long	Необработанное число тактов физического процессора в пользовательском режиме.
psys	unsigned long long	Необработанное число тактов физического процессора в системном режиме.
pidle	unsigned long long	Необработанное число тактов физического процессора в режиме простоя.
pwait	unsigned long long	Необработанное число тактов физического процессора в режиме ожидания операций ввода-вывода.
decrintrs	unsigned long long	Число прерываний декрементного счетчика.
mpcrintrs	unsigned long long	Число прерываний *Multi Processor Communication (MPC) receive*.
mpcsintrs	unsigned long long	Число прерываний *MPC send*.
phantintrs	unsigned long long	Число фантомных прерываний, полученных разделом.
idle_donated_purr	unsigned long long	Число циклов простоя, передаваемых выделенным разделом, в котором разрешена такая передача.
idle_donated_spurr	unsigned long long	Число циклов простоя SPURR, передаваемых выделенным разделом, в котором разрешена такая передача.
busy_donated_purr	unsigned long long	Число занятых циклов, передаваемых выделенным разделом, в котором разрешена такая передача.
busy_donated_spurr	unsigned long long	Число занятых циклов SPURR, передаваемых выделенным разделом, в котором разрешена такая передача.
idle_stolen_purr	unsigned long long	Число циклов простоя, забранных гипервизором у выделенного раздела.

Таблица 20. Поддерживаемые поля ProbeVue для общей статистики процессоров (продолжение)

Имя поля сценария Vue	Тип данных	Описание
idle_stolen_spurr	unsigned long long	Число циклов простоя SPURR, забранных гипервизором у выделенного раздела.
busy_stolen_purr	unsigned long long	Число занятых циклов, забранных гипервизором у выделенного раздела.
busy_stolen_spurr	unsigned long long	Число занятых циклов SPURR, забранных гипервизором у выделенного раздела.
iowait	short	Число спящих процессов, ожидающих буферизованных операций ввода-вывода.
physio	short	Число процессов, ожидающих низкоуровневых операций ввода-вывода.
twait	unsigned long long	Число нитей, ожидающих прямого ввода-вывода (DIO) или параллельного ввода-вывода (CIO) файловой системы.
hpi	unsigned long long	Число операций подкачки в оперативную память гипервизора.
puser_spurr	unsigned long long	Число циклов SPURR в пользовательском режиме.
psys_spurr	unsigned long long	Число циклов SPURR в режиме ядра.
pidle_spurr	unsigned long long	Число циклов SPURR в режиме простоя.
pwait_spurr	unsigned long long	Число циклов SPURR в режиме ожидания.
spurrflag	int	Это поле указывает, работает ли процессор в режиме SPURR.
tb_last	unsigned long long	Последнее прочитанное значение регистра timebase.

Примечание: Следующие элементы требуют доступа к памяти без закрепления: `devintrs`, `softintrs`, `loadavg1`, `loadavg5`, `loadavg15`, `decrintrs`, `mpcrintrs`, `mpcsintrs`, `phantintrs`. Поскольку тесты сценария Vue выполняются в среде, где все прерывания выключены, то в случае отсутствия необходимых данных в резидентной памяти элементам, которым требуется доступ к памяти без прикрепления, присваивается значение 0.

Сбор системной статистики

ProbeVue - это долговыполняемая команда и сценарий Vue запускаются несколько раз. Сценарий Vue может обращаться к устройствам или ресурсам для отображения статистики; в результате ресурсы и устройства могут стать временно или постоянно недоступными. Таким образом, операторы статистики системы ProbeVue в разных условиях работают следующим образом:

Таблица 21. Условия и варианты поведения статистики системы ProbeVue

Серийный номер	Условия	Поведение
1	Ресурс, указанный в операторе, не существует.	Если в ходе проверки ProbeVue не найдет ресурс, указанный в сценарии Vue, то возникает ошибка компиляции.
2	Устройство или ресурс не открыто/активно.	• ProbeVue отображает нулевые значения для статистики. • Устройство может быть открыто для выполнения функции **ioctl()** с последующим закрытием.
3	Устройство или ресурс переходит из активного в неактивное состояние.	ProbeVue отображает нулевые значения или устаревшие данные.
4	Устройство закрывается во время выполнения.	Те же результаты, что и в случае перехода устройства в неактивное состояние.
5	Время работы API асинхронной выборки превышает интервал выборки, заданный пользователем.	ProbeVue не может гарантировать правильные результаты, если время выборки превышает интервал выборки.
6	Ресурс, указанный в сценарии Vue, удаляется операцией динамического распределения ресурсов LPAR во время выполнения.	ProbeVue отображает нулевые значения или устаревшие данные.
7	Вызов выборки статистики устройства блокируется из-за тайм-аута драйвера устройства.	ProbeVue не может гарантировать правильность результатов и пользователю могут быть показаны устаревшие данные.

Присваивание типа и значения

Классификация в предшествующем разделе является одним из способов просмотра переменных в сценарии Vue. Классы переменных можно рассмотреть с разных точек зрения, в частности, как порождены их значения.

Внешние переменные:

Переменные класса ядра, переменные класса выхода и входа, а также встроенные переменные являются внешними переменными.

Они существуют независимо от структуры ProbeVue и порождают свои значения вне контекста любого сценария Vue. ProbeVue позволяет текущим значениям внешних переменных быть доступными внутри сценария Vue. Эти переменные всегда доступны только для чтения в контексте сценария Vue. Все операторы программы, которые пытаются изменить значение внешней переменной, будут помечены компилятором как недопустимые.

Хотя внешние переменные имеют предопределенный тип, ProbeVue требует явного объявления всех внешних переменных, кроме встроенных, в сценарии Vue, который к ним обращается. Следующая таблица описывает, как определены типы внешних переменных:

Переменная	Тип
Глобальный класс ядра	Их оператора объявления **__kernel** переменной ядра.
Класс входа	Из объявления прототипа функции в сценарии Vue. Необходимо указать типы данных каждого аргумента, используемого в сценарии Vue.
Значение возврата из функций ядра	Из объявления прототипа функции в сценарии Vue. Необходимо задать тип значения возврата.
Встроенные	Они обычно зависят от лежащей в основе переменной ядра. Ниже приведены их определенные типы и эквивалентные типы ProbeVue:

Встроенный	Определенный тип	Тип ProbeVue
__tid	tid_t	long long
__pid	pid_t	long long
__ppid	pid_t	long long
__pgid	pid_t	long long
__pname	char [32]	String [32]
__uid	uid_t	unsigned int
__euid	uid_t	unsigned int
__trcid	pid_t	long long
__errno	int	int
__kernelmode	int	int
__r3..__r10	32-разрядный для 32-разрядных процессов 64-разрядный для 64-разрядных процессов	unsigned long
__curthread	нет	Все значения имеют тип long long
__curproc	нет	Все значения, кроме **cwd**, имеют тип long long. Переменная **cwd** имеет тип string.
__ublock	нет	Все значения имеют тип long long

Примечание: Максимальный размер возвращаемых данных может быть меньше размера типа. Например, ИД процесса в AIX может вместиться в 32-разрядное целое число, в то время как тип данных pid_t является 64-разрядным целым числом для 64-разрядных процессов и ядра.

Переменные сценария:

Переменная сценария - это автоматическая переменная, переменная локальной нити или переменная глобального класса.

Переменные сценария существуют только внутри контекста сценария Vue, и их значения присваиваются из сценария. Кроме того, они могут быть присвоены или изменены только внутри сценария, который их определяет.

В общем случае, необходимо явно объявить тип данных переменной сценария посредством оператора объявления. Однако, компилятор может неявно определить тип данных программной переменной в некоторых ограниченных случаях, если первая ссылка на переменную находится в левой части оператора присваивания переменной.

Неявное определение типа для целых типов

Для того чтобы присвоить целый тип, правая часть оператора присваивания должна быть:

- Числовой константой.
- Другой переменной целого типа, включая встроенные переменные. Присваивание из переменной, тип которой неизвестен, является ошибкой.
- Функцией Vue, которая возвращает целый тип, такой как функция **diff_time**.
- Выражением преобразования типов в целый тип в правой части оператора, хотя это может вызвать предупреждение в некоторых случаях.
- Выражением, включающим в себя все указанные выше случаи.

Переменная принимает свой тип в дополнение к своему значению на основании выражения в правой части оператора. Кроме того, класс переменной может быть присвоен ей с помощью префикса. Следующий сценарий демонстрирует некоторые примеры:

```
/*
 * Файл: implicit2.e
 * Формат: Демонстрирует неявное присваивание для целых типов
 */

int read(int fd, char *p, long size);

@@BEGIN
{
 count = 404;   /* count: int глобального класса */
 zcount = 2 * (count - 4); /* zcount: int глобального класса */
 llcount = 33459182089021LL;   /* lcount: long long глобального класса */
 lxcount = 0xF00000000245B20LL; /* xcount: long long глобального класса */

}

@@syscall:$1:read:entry
{
 __auto probev_timestamp_t ts1, ts2;
 int gsize;
 ts1 = timestamp();
 auto:dcount = llcount - lxcount;  /* dcount: long long автоматического класса */

 auto:mypid = __pid; /* mypid:  pid_t (64-bit integer) автоматического класса */
 fd = __arg1;  /* fd: int глобального класса */

 /* Следующий оператор возможно вызовет предупреждение компилятора,
  * но здесь его можно проигнорировать
  */
 global:bufaddr = (long)__arg2; /* bufaddr: long глобального класса */

 gsize = __arg3;
```

```
 thread:size =  gsize + 400; /* size: int класса локальной нити */

 printf("count = %d, zcount = %lld\n", count, zcount);
 printf("llcount = %lld, lxcount = 0x%016llx, diff = %lld\n",
   llcount, lxcount, dcount);
 printf("mypid = %ld, fd = %d, size = %d\n", mypid, fd, size);
 printf("bufaddr = 0x%08x\n", bufaddr);
 ts2 = timestamp();

 auto:diff = diff_time(ts1, ts2, MICROSECONDS); /* diff: int автоматического класса */

 printf("Time to execute = %d microseconds\n", diff);

 exit();
}
```

Примечание: В приведенном выше сценарии существует позиционный параметр оболочки, а именно символ **$1** в спецификации теста **@@syscall:$1:read:entry**. Администратор тестирования **syscall** разрешает ИД процесса второго поля указывать, что точка тестирования системного вызова должна быть включена только для определенного процесса. В отличие от жестко заданного определенного ИД процесса, второе поле установлено в позиционный параметр оболочки в этом сценарии, чтобы позволить передачу фактического ИД процесса в качестве аргумента во время выполнения сценария. Команда **probevue** заменяет все позиционные параметры оболочки в сценарии на соответствующие аргументы, переданные в командной строке.

Предполагая, что тестируется процесс с ИД 250000, следующий сценарий показывает выполнение **implicit2.e**.

```
# probevue implicit2.e 250000
WRN-100: Line:29 Column:26 Incompatible cast
count = 404, zcount = 800
llcount = 33459182089021, lxcount = 0x0f00000000245b20, diff = -1080830451389212643
mypid = 250000, fd = 10, size = 4496
bufaddr = 0x20033c00
Time to execute = 11 microseconds
```

В этом примере символ **$1** в сценарии автоматически заменяется на "250000", ограничивая тем самым точку теста входа системного вызова процессом с ИД, равным 250000.

Неявное определение типа для строкового типа

Для того чтобы присвоить строковый тип, правая часть оператора присваивания должна быть:

- Строковым литералом, то есть последовательностью символов в двойных кавычках.
- Другой переменной строкового типа, включая встроенные переменные.
- Функцией Vue, которая возвращает строку, например, функция **et_userstring**.
- Выражением, включающим в себя все указанные выше случаи.

Следующий пример демонстрирует неявное присваивание строкового типа:

```
/*
 * Файл: implicit3.e
 * Формат: Демонстрирует неявное присваивание для строковых типов
 */

int write(int fd, char *p, long size);

@@BEGIN
{
 s1 = "Write system call:\n";
}

@@syscall:$1:write:entry
{
```

```
 String s2[40];

 wbuf = get_userstring(__arg2, __arg3);

 s2 = s1;

 zbuf = s2;

 pstring = zbuf + wbuf;

 printf("%s\n", pstring);
}

@@syscall:$1:write:exit
{
 ename = __pname;
 printf("Exec name = %s\n", ename);
 exit();
}
```

ИД процесса должен быть передан в качестве аргумента в сценарий при замене переменной позиционного параметра оболочки **$1**.

Неявное определение типа списка

Для того чтобы присвоить тип списка, правая часть оператора присваивания должна быть функцией **list()**. Функция **list()** поддерживается из любого оператора.

Модели данных для 32-разрядных и 64-разрядных процессов:

AIX поддерживает две среды разработки: 32-разрядную и 64-разрядную.

Компиляторы в AIX предлагают две следующие программные модели:

ILP32 ILP32 (акроним для integer, long и pointer 32) - это 32-разрядная среда программирования в AIX. Модель данных ILP32 предоставляет 32-разрядное адресное пространство с теоретическим ограничением памяти, равным 4 ГБ.

LP64 LP64 (акроним для long и pointer 64) - это 64-разрядная среда программирования в AIX. За исключением типа данных и выравнивания, LP64 поддерживает те же функции программирования, что и модель ILP32, и обратно совместима с наиболее широко используемым типом данных int.

Таким образом, программа в AIX может быть скомпилирована для выполнения как 32-разрядная программа или как 64-разрядная программа. Один и тот же сценарий Vue может быть выполнен для процессов, выполняющихся в 32-разрядном или в 64-разрядном режиме. Что касается спецификации модели данных, внешняя переменная типа long, доступная в сценарии Vue, должна считаться имеющей длину 4 байта при тестировании (или трассировке) 32-разрядного процесса. Та же переменная должна считаться имеющей длину 8 байтов при тестировании 64-разрядного процесса. Макет и размер структуры или объединения, содержащих элементы, которые являются указателями или переменными long, будут зависеть от того, видима ли она из 32-разрядного процесса или из 64-разрядного процесса. Во избежание путаницы, Vue предоставляет семантические правила для управления двумя различными моделями данных в логической или согласованной манере на основании класса переменной.

Типы переменных постоянного размера:

`int`, `short`, `char` и `long long` - это типы переменных постоянного размера.

Они всегда имеют одинаковый размер как в 32-разрядном, так и в 64-разрядном режиме, независимо от класса объявленной переменной.

Тип	Размер
long long	8
int	4
short	2
char	1

Типы переменных переменного размера:

Типы переменных переменного размера могут использоваться как в 32-разрядном, так и в 64-разрядном режиме.

Типы данных переменного размера перечислены ниже:

Тип	Размер в 32-разрядном режиме	Размер в 64-разрядном режиме
long	4	8
Типы указателей	4	8

В приведенной выше таблице тип указателя ссылается на такие типы, как **char ***, **int ***, **struct foo ***, **unsigned long *** и так далее.

Следующие семантические правила применяются к переменным, которые определены с помощью одного из указанных выше типов, то есть для "long" и "pointer". Правила применяются независимо от того, являются ли переменные элементами структуры или объединения, или они объявлены как отдельные переменные:

Автоматический класс
Режим переменной будет зависеть от режима тестируемого процесса (32 или 64).

Класс локальной нити
Режим переменной будет зависеть от режима тестируемого процесса (32 или 64).

Глобальный класс
Переменная всегда обрабатывается в 64-разрядном режиме, независимо от режима тестируемого процесса. Это позволяет безопасно использовать переменную как в 32-разрядных, так и в 64-разрядных процессах без потери данных.

Глобальный класс ядра
Переменные ядра, которые являются указателями или имеют тип long, всегда обрабатываются в 64-разрядном режиме, как единственном поддерживаемом ядре для AIX 6.1.

Класс входа
Если тип long или pointer определен в прототипе функции для любого из параметров функции, режимы соответствующих переменных класса входа (от **__arg1** до **__arg32**) будут зависеть от режима тестируемого процесса (32 или 64).

Класс выхода
Если тип long или pointer определен в прототипе функции как тип значения возврата функции, режимы переменной класса выхода (**__rv**) будут зависеть от режима тестируемого процесса (32 или 64).

Встроенный класс
Эти переменные обычно имеют тип постоянного размера, кроме встроенных переменных от **__r3** до **__r10**, которые определены с помощью типа long без знака и имею длину 32 бита для 32-разрядных процессов и 64 бита для 64-разрядных процессов.

Тесты **@@BEGIN** и **@@END** всегда выполняются в 64-разрядном режиме.

Типы данных в Vue

Язык Vue принимает три специальных типа данных в дополнение к обычным типам данных C-89.

Типы данных, порожденные из языка C

Типы данных, порожденные из языка C

Язык Vue поддерживает большинство типов данных, определенных в спецификации C-89. Они включают в себя версии целых типов данных со знаками и без знаков: **char**, **short**, **int**, **long** и **long long**. "plain" **char** считается не имеющим знака, в то время как другие целые типы, если они не объявлены явно, считаются имеющими знак. Это соответствует реализации C в PowerPC. Язык Vue также поддерживает типы с плавающей точкой: **float** и **double**. В дополнение к этим основным типам языка C, Vue также поддерживает производные типы, такие как массив, структура, объединение, указатель, перечисление и некоторые неполные типы, например, **void**.

Типы с плавающей точкой
: Тип с плавающей точкой можно использовать только в простых выражениях присваивания и в качестве аргументов для таких функций, как, например, **printf**. В частности, невозможно использовать переменные с плавающей точкой как операнды любых унарных и бинарных операторов, кроме оператора присваивания.

Типы указателей
: Указатели можно использовать для раскрытия ссылки на данные приложения или ядра. Однако, нельзя объявить указатели на переменные сценария Vue или взять их адреса.

Массивы символов
: Невозможно использовать массив символов как строку в C, для чего используется строковый тип данных.

Неполные типы
: Невозможно использовать типы массива неизвестного размера.

Типы битового поля
: Компилятор Vue игнорирует объявления битового поля и макет структуры или объединения, содержащих элементы битовых полей, не определен.

Модели данных ILP32 и LP64
: Обычно, программа C может быть скомпилирована в 32-разрядном режиме, где она соответствует модели данных ILP32, или в 64-разрядном режиме, где она соответствует модели LP64. Так как один и тот же оператор Vue может быть выполнен как в 32-разрядных, так и в 64-разрядных процессах, Vue поддерживает обе модели одновременно.

Тип данных Range и Bucket

Тип данных Range и Bucket

Тип данных range в Vue предназначен для обработки распределения точек данных в заданных диапазонах. В каждом диапазоне подсчитывается число попадающих в него элементов. Допустимы целочисленные и строковые диапазоны. Поддерживаются линейное и квадратичное распределения значений диапазонов. Примеры линейного и квадратичного распределений значений диапазонов:

Линейное распределение:

Диапазон	Число
0 - 5	2
5 - 10	4
10 - 15	1
Другие	20

Квадратичное распределение

Диапазон	Число
1 - 2	61 см
2 - 4	9
4 - 8	2005
8 - 16	4
16 - 32	1999
32 - 64	7
Другие	5

В предыдущих примерах распределений показано число элементов, значения которых лежат в заданных диапазонах. Например, в квадратичном распределении в диапазон 4 - 8 попадает 2005 элементов данных. Элементы, значения которых не лежат в заданных диапазонах, отображаются в категории Другие.

Пример строковых диапазонов

Пример:

Диапазон	Число
Read, write, open	87
Close, foo1	3
foo2	1
Другие	51

В предыдущем примере распределение указывает число вхождений конкретной строки в указанные значения диапазонов. В этом примере read, write и open вызывались 87 раз.

Объявление и инициализация диапазона:

Тип данных range можно объявить с помощью ключевого слова `range_t`. Например следующее объявление в сценарии Vue задает две переменные, содержащие диапазоны:

```
range_t T1, T2;  // T1 и T2 - это переменные типа Range.
```

Процедуры `set_range` и `add_range` применяются для инициализации целочисленных и строковых диапазонов в любых переменных типа Range.

Инициализация целочисленного диапазона: для инициализации целочисленных диапазонов применяется процедура `set_range`. Синтаксис процедуры `set_range` зависит от типа распределения значений диапазона: линейное или квадратичное. Синтаксис процедуры `set_range` для линейного распределения:

```
void set_range(range_t range_data, LINEAR, int min, int max, int step);
```

Пример:

```
set_range(T1, LINEAR, 0, 100, 10);
```

В предыдущем примере процедура `set_range` выполняет инициализацию данных диапазона T1 с линейным распределением. Нижняя граница T1: 0, верхняя граница: 100. Размер каждого диапазона: 10. Соответствующее распределение будет выглядеть следующим образом:

Диапазон	Число
0 - 10	...
10 - 20	...
20 - 30	...
...	...
...	...
90 - 100	...

Для инициализации квадратичного распределения применяется следующий синтаксис:

```
set_range(range_t range_data, POWER, 2);
```

Пример:

```
set_range(T2, POWER, 2);
```

В этом примере процедура выполняет инициализацию диапазона T2 с квадратичным распределением.

Инициализация строкового диапазона: для инициализации строковых диапазонов применяется процедура `add_range`.

Формат:

```
void add_range(range_t range_data , String S1, String S2, ..., String Sn);
```

Пример:

```
add_range(T1, "read", "write", "open");
```

Эта процедура добавляет строки `read`, `write` и `open` в одну ячейку `range_t` T1. Другая процедура **add_range** для `range_t` T1 добавит строки в следующую ячейку.

```
add_range(T1, "close", "func1", "func2");
```

Эта процедура добавляет строки `close`, `func1` и `func2` в следующую ячейку объекта `range_t` T1.

Примечание: range_t - это специальный тип данных Vue, который можно сохранить в качестве значения только в именованном массиве. Другие операции (например арифметические, логические, побитовые, реляционные) над типом данных **range_t** не поддерживаются.

Notes: Рассмотрены различные способы применения типа данных **range_t** и процедуры его инициализации.

1. Тип данных **range_t** можно объявить только в предложении @@BEGIN.
2. Функцию инициализации **set_range** можно использовать только в предложении @@BEGIN.
3. Диапазон с целочисленными значениями можно инициализировать только один раз. Одну и ту же переменную нельзя инициализировать два раза.

   Пример:

   ```
   set_range(T1, LINEAR, 0, 50, 5); // Допустимый синтаксис
       set_range(T1, LINERA, 10, 100, 10); // Ошибка, невозможно выполнить инициализацию, поскольку
                                           // T1 уже инициализирован.
       set_range(T1, POWER, 2);            // Ошибка, T1 уже инициализирован.
       add_range(T1, "read", "write");     // Ошибка, T1 уже инициализирован.
   ```
4. Параметры `min`, `max` и `step` должны быть целочисленными константами процедуры **set_range**.

Хранение и печать диапазонов:

Диапазон можно сохранить в именованном массиве в качестве значения с помощью процедуры `qrange`. Функция `qrange` находит номер ячейки, счетчик которой требуется увеличить.

Пример:

В этом примере T1 - это объект типа **range_t** с целочисленными значениями.

```
qrange(aso["read"], T1, time_spent);
```

В этом примере процедура `qrange` находит номер ячейки, к которой относится *time_spent*, и увеличивает счетчик этой ячейки в именованном массиве `aso` с учетом ключа read.

В следующем примере T2 - это объект типа **range_t** с строковыми значениями.

```
qrange(aso["function usage"], T2,get_function());
```

В этом примере процедура `qrange` находит номер ячейки, к которой относится функция, переданная в качестве третьего аргумента; счетчик для этой ячейки увеличивается в именованном массиве `aso` с учетом ключа **function usage**.

Notes:

1. В любом ASO в качестве значения можно сохранить только одну переменную типа **range_t**. Функцию **qrange** нельзя использовать для переменных **range_t** разных типов в одном ASO.

   Пример:

   ```
   qrange(aso["read"], T1,time_spent); // Правильный синтаксис.
   qrange(aso["read"], T2,time_spent); // Ошибка. Две переменные range_t разных типов
                                       // нельзя использовать в одном ASO.
   ```

   Функции `quantize` и `lquantize` именованного массива со значениями типа **range_t** отображают визуальное представление счетчиков диапазонов.
2. В случае строкового диапазона в одной ячейке можно напечатать не более 40 символов (включая запятые). Если ячейка содержит более 40 символов, то строковый диапазон усекается - при выводе последние 3 символа заменяются точками (...).

Примеры диапазонов и процедуры qrange:

```
@@BEGIN
{
  __thread start ;
  range_t T1;
  set_range(T1, LINEAR, 0, 150, 10) ;
}
@@syscall :$__CPID :read :entry
{
  thread :tracing = 1 ;
  start = timestamp() ;
}
@@syscall :$__CPID :read :exit
         when(thread :tracing == 1)
{
  __auto long time_spent;
  currtime = timestamp() ;
  time_spent = diff_time(start, currtime, MICROSECONDS);
  qrange(aso["read"], T1, time_spent);
}
@@END
{
  print(aso);
  quantize(aso);
}
```

Вывод этого примера выглядит следующим образом:

```
Key                                     Value

Read            Range                   count
                0-11                    4
                10-20 6
                60-70 7
```

```
                Others                  32

Key                                     Value

Read            Range                   count
                0-10 4 ===
                10-20 6 ====
                60-70 7 =====
                Others                  32      ================
```

Тип трассировки стека

Тип трассировки стека

Значение, возвращаемое функцией ProbeVue **get_stktrace** и содержащее трассировку текущего стека, имеет тип **stktrace_t**. Это трассировка стека текущей нити. Переменную с трассировкой можно хранить в именованном массиве как ключ или как значение. Тип **stktrace_t** - это абстрактный тип данных. Переменную такого типа нельзя использовать напрямую в стандартных унарных или бинарных операторах C. Во внутреннем представлении такая переменная являет собой массив значений unsigned long.

Vue поддерживает следующие свойства и операции для переменных типа "трассировка стека":

Объявление переменной типа "трассировка стека"

Переменную можно объявить следующим образом:

```
stktrace_t  st;              // st - переменная типа stktrace_t.
st = get_stktrace(5);        //  Получить трассировку до пятого уровня.
a_st[0] = get_stktrace(-1);  // Получить всю доступную трассировку и
                             // сохранить ее в именованном массиве a_st как одно значение.
```

Переменные типа **stktrace_t** не поддерживают спецификаторы signed, unsigned, register, static, auto, thread, kernel и const.

Операции присвоения

Оператор присвоения **(=)** позволяет присвоить одной переменной типа **stktrace_t** значение другой переменной типа **stktrace_t**. Исходное значение переменной **stktrace_t**, которой присваивается значение, будет утеряно. Преобразования типов между переменными типа **stktrace_t** и других типов не допускаются. В следующем примере содержимое трассировки стека *t1* присваивается переменной *t2*.

```
stktrace_t     t1, t2;          // Объявление двух переменных типа "трассировка стека".
t1 = get_stktrace();            // Запись трассировки текущего стека в t1.
t2 = t1 ;                       // Копирование содержимого t1 в переменную t2.
```

Операция сравнения

Только операции проверки равенства **(==)** и неравенства **(! =)** допустимы между переменными типа **stktrace_t**. Результатом этих операторов будет либо True(1), либо False(0), в зависимости от полных значений переменных **stktrace_t**. Сравнение отдельных записей стека **stktrace_t** недопустимо. Другие операции сравнения (**>=**,**>**, **<** or **=**<) для типа **stktrace_t** не поддерживаются.

```
          if( t1 == t2)                 // сравнение двух переменных типа stktrace_t.
                  printf("Записи равны");
            else
                  printf("Записи не равны");
```

Печать переменной типа "трассировка стека"

| Для вывода значения типа **stktrace_t** используется функция Vue **printf** со спецификаторами формата **%t** или
| **%T**. Трассировка стека будет выведена в символьном формате. Символы адресов (символ плюс адрес)

| печатаются только в том случае, если нить, трассировка стека которой содержится в переменной **stktrace_t**,
| выполняется в данный момент и для печати трассировки стека задан спецификатор **%t**; иначе печатаются
| только адреса.

Переменные типа **stktrace_t** хранятся в именованном массиве в качестве ключа или значения и могут быть напечатаны с помощью функций print такого массива. В если в переменной **stktrace_t** хранится стек нити, которая выполняется на текущий момент, то адреса будут напечатаны с символами, иначе будут напечатаны
| только значения адресов. Если задан флаг **STKTRC_NO_SYM** для функции **set_aso_print_options()**, то эта
| переменная содержит непреобразованные адреса работающей нити.

```
stktrace_t t1;
t1 = get_stktrace (5);
printf ("%t", t1);       // Показывает трассировку стека из переменной t1.
printf ("%T", t1);      // Показывает трассировку стека из переменной t1 с непреобразованными адресами.
a[__tid] = t1;           // Сохраняет t1 как элемент именованного массива a.
print(a) ;              // Печатает именованный массив
               // с элементами типа stktrace_t.
```

Ограничения переменных типа "трассировка стека"

- Нельзя объявить массив переменных **stktrace_t**.
- Переменные **stktrace_t** не могут быть часть структуры или объединения.
- Запрещен доступ к отдельным элементам трассировки стека.
- Операции (присвоение, сравнение и печать) над переменными типа **stktrace_t** не поддерживаются в тестах systrace.

Специальные типы данных

В дополнение к обычным типам данных C-89 язык Vue также принимает семь специальных типов данных.

Строковый тип:

Строковый тип данных - это представление строковых литералов. В отличие от C, строка - это базовый тип данных в Vue

Наличие строкового типа позволяет избежать некоторых конфликтов в C, где не поддерживается строковый тип, но разрешается представление строки как указателем на тип **char**, так и массивом символов. Можно явно объявить строковую переменную с помощью оператора объявления строки. Явно объявленные строковые переменные должны также указывать максимальную длину строки (подобно объявлению массива символов в C). В отличие от C, строка в Vue явно не завершается нулевым символом, и вам не обязательно резервировать для него пространство.

```
String s[40]; /* Определяет строку 's' длиной 40 */
s = "probevue";
```

Далее, любому строковому литералу, записанному в стиле C в двойных кавычках, автоматически назначается строковый тип. Vue автоматически преобразовывает внешнюю переменную, объявленную как символьный тип данных в стиле C (**char *** или **char[]**), в строковый тип данных, если необходимо.

Для строкового типа данных можно использовать следующие операторы:

- Оператор конкатенации: "+" .
- Оператор присваивания: "=" .
- Относительные операторы для сравнения строк: "==", "!=", ">", ">=", "<" и "<=".

Можно установить строковую переменную в пустую строку, присвоив ей значение "", как показано в следующем примере:

```
s = "";  /* Устанавливает s как пустую строку */
```

В отличие от C, для пары смежных строковых литералов конкатенация не совершается автоматически. Оператор конкатенации (+) должен быть явно применен, как в следующем примере:

```
String s[12];

 // s = "abc" "def";
   /* ОШИБКА: Закомментированный выше оператор приведет к синтаксической ошибке */
 s = "abc" + "def"; /* Правильный способ конкатенации строк */
```

Vue поддерживает несколько функций, которые принимают строковый тип данных как параметр или возвращают значение, имеющее строковый тип данных.

Списочный тип:

Переменные списочного типа собирают набор целочисленных значений. Списочный тип - это абстрактный тип данных. Переменную списочного типа нельзя использовать напрямую с унарными или бинарными операторами C.

Для списочного типа данных можно использовать следующие операции:

- Функция конструктора **list()** для создания новой переменной списка, если она не объявлена ранее. Если переменная уже определена, она должна быть очищена.
- Функция конкатенации **append** для добавления элемента к списку или соединения двух списков.
- Оператор "=", позволяющий назначать один список другому.
- Набор функций агрегирования, которые действуют на переменной списка и возвращают скалярное (целое) значение: **sum**, **avg**, **min**, **max** и так далее.

Хотя можно использовать переменную списка для сбора целого значения, значения всегда сохраняются как 64-разрядные целые числа со знаком.

Функция **list()** возвращает новый пустой список, который должен быть присвоен переменной типа списка. Это создаст новую переменную списка, если переменная списка в левой части оператора присваивания еще не была присвоена списку. Она также может быть присвоена существующей переменной списка, и в этом случае все значения, собранные в целевом списке, отбрасываются. Далее, переменная может быть объявлена как переменная типа списка в любом месте сценария Vue следующим образом:

```
__list l_opens;
```

Эффект этого равносилен вызову функции **list()** в тесте @@BEGIN, а возвращенное значение присваивается этой переменной списка.

Следующий пример создает списочную переменную с именем **l_opens**:

```
l_opens = list();
```

Функцию **list** можно вызывать из любого блока. Если в функцию **list** передается имя существующего списка, из этого списка удаляются все элементы.

Функцию **append()** можно использовать для добавления значения в переменную списка. Каждый вызов функции **append** добавляет новое значение к набору значений, уже сохраненных в переменной списка. Следующий пример показывает, как Размер переменной списка вырастает при каждом вызове функции **append**:

```
append(l_opens, n_opens1); /* l_opens = {n_opens1} */
append(l_opens, n_opens2); /* l_opens = {n_opens1, n_opens2} */
append(l_opens, n_opens3); /* l_opens = {n_opens1, n_opens2, n_opens3} */
append(l_opens, n_opens4); /* l_opens = {n_opens1, n_opens2, n_opens3, n_opens4} */
```

Второй параметр функции **append()** может быть также переменной списка, все значения которого будут добавлены в целевой список, указанный в первом параметре. Таким образом, функция append может быть использована для соединения двух списков.

В следующем примере содержимое списка b добавляется к списку a:

```
a=list()
b=list()
append(a,b)
```

Примечание: Добавленное в список значение должно иметь тип целого или списка, и в случае, если какая-либо из переменных **n_opens1 -n_opens4** не будет иметь целый тип, возникнет ошибка. Любые типы, которые меньше, чем **long long** (такие как short или int), автоматически преобразовываются в тип **long long**.

Функцию append можно также использовать для соединения двух списков. Первый аргумент - это целевой список, а второй - это исходный список. В следующем примере содержимое списка b добавляется к списку a:

```
a=list()
b=list()
append(a,b)
```

Функция **append()** не имеет возвращаемого значения.

Список можно присвоить другому списку с помощью оператора присвоения. Исходные значения в целевом списке уничтожаются. В следующем примере содержимое **l_opens2** теряется (элементы удаляются), а содержимое списка **l_opens** записывается поверх списка **l_opens2**.

```
l_opens2 = list();
append(l_opens2, n_opens5);

l_opens2 = l_opens;
  /* l_opens and l_opens2 => {n_opens1, n_opens2, n_opens3, n_opens4}  */
```

Функция агрегирования может быть применена к переменной списка, как показано в следующем примере:

```
/* ниже мы предполагаем, что n_opens1=4, n_opens2=6, n_opens3=2 и n_opens4 = 4
 * они были добавлены к переменной списка l_opens
 */
x = avg(l_opens); /* задаем для x значение 4 */
y = min(l_opens); /* задаем для y значение 2 */
z = sum(l_opens); /* задаем для z значение 16 */
a = count(l_opens) /* задаем для a значение 4 */
```

Переменная списка полезна, когда необходимо записать точные агрегатные значения. Переменные списка обновляются автоматически, поэтому применяйте их только, когда это требуется, так как они менее эффективны, чем обычные переменные.

Именованный массив:

Именованный массив - это тип данных, представляющий собой набор ключей и соответствующих им значений. Между ключами и значениями соблюдается связь один к одному. Именованные массивы поддерживаются в Perl, ksh93 и других языках.

В Vue поддерживаются только связи из одного или нескольких ключей с одним значением. Именованные массивы могут иметь ключи следующих типов:

- integral
- floating point
- string
- timestamp
- stacktrace
- path
- MAC address
- IP address

Именованные массивы могут иметь значения следующих типов:

- integral
- floating point
- string
- timestamp
- stacktrace
- path
- MAC address
- IP address
- list
- range

Именованные массивы относятся к абстрактным типам данных в Vue.Следующие действия можно выполнить над типами данных именованных массивов.

- Привязка ключей и значений:

  Если экземпляр ключей еще не существует, то это действие добавляет ключ или набор ключей со связанным значением в именованный массив. В противном случае это действие заменяет старое значение новым значением. По умолчанию ключам присваивается значение 0 для типа `numerical`, пустая строка для типа `string` или значение `NULL` для других типов.

  В следующем примере демонстрируется привязка ключей к значениям

  ```
  /* один ключ */
  count["ksh"] = 1;
  /* несколько ключей */
  var[0]["a"][2.5] = 1;
  var[1]["a"][3.5] = 2;
  ```

  При первом использовании переменной именованного массива указывается тип ключей, размер индексов и тип значения. Они должны применяться ко всему сценарию Vue.

  Для ключа в ASO можно привязать значение LIST, выполнив следующие действия:

  1. Путем назначения переменной LIST:

     ```
     assoc_array["ksh"]=ll /* список ll копируется в именованный массив */
     assoc_array["ksh"]=assoc_array["abc"]; /* список в ASO копируется в другой список в ASO.
                                               Массив assoc_array содержит значения типа LIST */
     ```

  2. Путем назначения пустого списка, возвращаемого конструктором **list()**:

     ```
     assoc_array["ksh"]=list(); /* присваивается пустой список */
     ```

  3. Путем добавления списка или целочисленного значения

     ```
     append(assoc_array["ksh"], 5); /* целое значение 5 добавляется в список в ASO */
     append(assoc_array["ksh"], ll); /* содержимое переменной LIST добавляется в список в ASO*/
     append(assoc_array["ksh"], assoc_array["abc"]); /* содержимое списка в ASO добавляется в другой список в ASO */
     ```

- Аннулирование связи ключа или набора ключей и удаление связанного значения: функция **delete()** используется для удаления ключей и их значений из именованного массива. Ключ с аннулированной связью имеет значение 0 или пустая строка.

  В следующем примере показано, как отменить привязку ключа с помощью функции удаления

  ```
  delete(count, "ksh");
  delete(var, 0, "a", 2.5);
  ```

  В качестве первого аргумента указывается имя переменной именованного массива. За именем переменной массива должны следовать N разделенных запятыми ключей, где N - это размерность массива. Если требуется удалить связанное значение для индексов, отличных от N, укажите `ANY` для этих индексов. Например, для удаления всех элементов со строкой “a” в качестве второго параметра введите следующую команду.

  ```
  delete(var, ANY, "a", ANY);
  ```

Если в функции delete() указать значение ANY для всех ключей, то будут удалены все элементы именованного массива. Эта функция возвращает 0, если указанные элементы найдены и удалены. В противном случае функция delete() возвращает значение 1.

- Поиск значения для набора ключей: эта операция выполняет поиск значений, связанных с одним или несколькими ключами.

```
total = count["ksh"] + count["csh"];
prod = var[0]["a"][2.5] * var[1]["a"][3.5];
```

Значение LIST для ключа можно извлечь путем индексации именованного массива с помощью ключа. К списку в именованном массиве применимы все функции LIST: sum(), min(), max(), count() и avg(). Кроме того, список в именованном массиве можно присвоить переменной LIST.

Пример:

```
/* копирование списка из именованного массива в переменную "ll" */
ll=assoc_array["ksh"];
/* печать суммы всех элементов списка из именованного массива, индексированного с помощью ksh" */
printf("sum of assoc_array %d\n",sum(assoc_array["ksh"]) );
/* печать минимального значения */
printf("min of assoc_array %d\n",min(assoc_array["ksh"]) );
/* печать максимального значения */
printf("max of assoc_array %d\n",max(assoc_array["ksh"]) );
/* печать числа значений в списке */
printf("count of assoc_array %d\n",count(assoc_array["ksh"]) );
/* печать среднего значения списка */
printf("avg of assoc_array %d\n",avg(assoc_array["ksh"]) );
```

- Проверка существования ключа или набора ключей: функция exists() позволяет проверить, содержит ли именованный массив элементы, соответствующие указанным ключам. Функция exists() возвращает значение 1, если элемент обнаружен; в противном случае возвращается значение 0.

Следующий фрагмент кода проверяет, существует ли ключ или набор ключей.

```
if (exists(count, "ksh"))
  printf("Число вызовов ksh = %d\n", count["ksh"]);
if (exists(var, 0, "a", 2.5))
  printf("Found value = %d\n", var[0]["a"][2.5]);
```

Если для конкретного индекса указано ключевое слово ANY, то он не учитывается в операции поиска. Для всех ключей в функции exists() можно указать значение ANY; в этом случае функция exists() проверит, содержит ли именованный массив любые элементы.

```
my_key = "a";
if (exists(var, ANY, my_key, ANY))
  printf("Found element with second key as %s \n", my_key);
```

- Операция увеличения или уменьшения: позволяет увеличить или уменьшить значение именованного массива. Для применения этой операции ключ должен содержать целочисленные значения. В следующих примерах показано применение операций увеличения и уменьшения:

1. printf(“Увеличенное значение = %d\n”, ++count[“ksh”]);
2. printf(“Увеличенное значение = %d\n”, count[“ksh”]++);
3. printf(“Уменьшенное значение = %d\n”, --count[“ksh”]);
4. printf (“Decremented value = %d\n”, count[“ksh”]--);

В примере 1 значение, соответствующее ключу **ksh**, сначала увеличивается, а затем отображается.

В примере 2 значение, соответствующее ключу **ksh**, сначала отображается, а затем увеличивается. Операция уменьшения работает аналогичным образом. Операции увеличения и уменьшения допустимы только для именованных массивов, содержащих значения типа integer. Эти операции можно также использовать для объединения, при этом для значения будет установлен тип данных integer. Например, если оператор a[100]++ встречается впервые, то будет создан именованный массив a с ключом типа integer и значением типа integer. Для ключа 100 будет задано значение 1. В случае выражения a[100]-- для ключа 100 будет задано значение -1. При последовательном использовании операций увеличения или уменьшения с одним и тем же массивом a эти операции применяются к указанному ключу.

В случае многомерных именованных массивов операции увеличения или уменьшения работают аналогичным образом:

```
++var[0]["a"][2.5];
var[0]["a"][2.5]++;
--var[1]["a"][3.5];
var[1]["a"][3.5]--;
```

- Вывод содержимого именованного массива: эта операция выводит ключ и связанное значение для элементов именованного массива. Можно указать следующие параметры печати:

Параметр печати именованного массива	Описание	Допустимые значения	Значение по умолчанию
num-of-entries	Позволяет напечатать первые пары ключ-значение.	*n*>=0. (Если указано значение 0, то отображаются все записи.)	0
sort-type	Задает порядок сортировки.	SORT_TYPE_ASCEND, SORT_TYPE_DESCEND	SORT_TYPE_ASCEND
sort-by	Задает критерий сортировки: ключ или значение.	SORT_BY_KEY, SORT_BY_VALUE	SORT_BY_KEY
list-value	Задает атрибут LIST для сортировки или квантования, если именованный массив содержит значение типа List	USE_LIST_SUM, USE_LIST_MIN, USE_LIST_MAX, USE_LIST_COUNT, USE_LIST_AVG	USE_LIST_AVG
sort-key-index	Задает индекс ключа для сортировки вывода.	-1 или k, где 0 <= k < число-ключей	0
stack-raw	Печать трассировки стека в формате адресов RAW.	STKTRC_NO_SYM	0

Если для флага sort-by указано значение SORT_BY_KEY, SORT_BY_VALUE, а пара ключ-значение не допускает сортировку, то *num-of-entries* и другие параметры печати используются для печати отдельной пары ключ-значение (если применимо). Например, в случае сортировки по типу диапазона параметр *num-of-entries* и другие параметры печати резервируются для ячеек каждого диапазона.

Параметры печати именованного массива по умолчанию можно изменить с помощью функции **set_aso_print_options()** в тесте BEGIN.

Пример:

```
set_aso_print_options (10, SORT_TYPE_DESCEND|SORT_BY_VALUE);
```

Обратите внимание, что с помощью вертикальной черты можно указать несколько флагов.

Примечание: Опцию sort-key-index нельзя указать с помощью функции set_aso_print_options(), поскольку ее нельзя обобщить для именованных массивов разных размерностей.

Функция print() печатает ключи и связанные значения всех элементов или подмножества элементов именованного массива с помощью параметров печати по умолчанию. Для переопределения параметров печати по умолчанию укажите дополнительные аргументы в функции print(). Дополнительная информация о функции print() приведена в разделе Функции Vue.

Функция **print()** печатает пары ключ-значение именованного массива с помощью параметров печати по умолчанию. Для просмотра содержимого именованного массива в другом формате флаг *num-of-entries* и параметры печати следует указать в качестве дополнительных параметров функции **print()**.

Пример:

```
/* отображение содержимого именованного массива 'count' с помощью параметров печати по умолчанию */
print(count);
/* печать первых 10 записей отсортированного именованного массива 'count'.
Применяются параметры sort-by и sort-type по умолчанию */
print(count, 10);
/* сортировка именованного массива 'count' в порядке убывания значений и
отображение первых 10 записей 'count' */
print(count, 10, SORT_BY_VALUE|SORT_TYPE_DESCEND);

/* печать элементов с первым ключом 0 */
print(var[0][ANY][ANY]);
```

- Процедура `clear()` позволяет сбросить ключи и связанные значения элементов именованного массива. Процедура `clear()` также позволяет сбросить значение связанного ключа массива без очистки ключей. Если процедура `clear()` успешно очищает один или несколько элементов, то она возвращает значение 0; в противном случае возвращается значение 1.

  ```
  clear(count);              // count - именованный массив.
  ```

  Функция с одним аргументом очищает все пары ключ-значение, существующие в именованном массиве **count**. После выполнения этой функции именованный массив **count** пуст.

  ```
  clear(count, RESET_VALUE);       // count - именованный массив.
  clear(var); // var - это трехмерный именованный массив
  ```

  Функция очищает все значения в парах ключ-значение, но не удаляет ключи. Значения в существующих парах ключ-значения после сброса зависят от их типа:

  Для очистки элементов с конкретными ключами необходимо указать ключи в первом аргументе. Кроме того, для того чтобы проигнорировать конкретный индекс ключа (все значения для конкретного индекса), можно указать значение `ANY`. Если все ключи указаны, то в качестве всех индексов именованного массива необходимо указать конкретные значения соответствующего типа или `ANY`.

  ```
  clear(var[ANY]["a"][ANY]);  // очистить все элементы,
  со вторым ключом "a"
  ```

  В процедуре `clear()` можно указать второй параметр `RESET_VALUE`. Если указано значение `RESET_VALUE`, то ключи именованного массива сохраняются и сбрасываются только значения.

  ```
  clear(count, RESET_VALUE);
  clear(var[0][ANY][ANY], RESET_VALUE);
  ```

  Параметр `RESET_VALUE` зависит от типа значения. В следующей таблице показаны типы данных вместе со связанными значения по умолчанию:

Тип	**Значение по умолчанию**
Целочисленные типы (int, long, short, long long)	0
LIST	Пустая строка
Число с плавающей точкой одинарной и двойной точности	0.0000000
Строка	Пустая строка
stktrace_t	Пустая строка
probev_timestamp_t	0
path_t	Пустая строка
mac_addr_t	0
ip_addr_t	0

- Операция `Quantize` печатает ключи и значения конкретного именованного массива в графическом формате с линейным масштабированием значений.

  ```
  quantize(count);
  ```

  count - именованный массив. Отображается следующая информация:

  ```
  ключ                значение
  1                   1                   ========
  2                   2                   =========
  3                   3                   ==========
  4                   4                   ===========
  5                   5                   ============
  6                   6                   =============
  ```

Параметры печати функции **quantize()** по умолчанию можно переопределить (аналогично функции **print()**).

Пример:

```
/* сортировка именованного массива 'count' в порядке убывания значений и
отображение первых 10 записей 'count' в графическом формате */
quantize(count, 10, _BY_VALUE|SORT_TYPE_DESCEND);
```

Для многомерных именованных массивов ключи можно указать в первом аргументе, чтобы указать конкретные элементы:

```
quantize(var[0][ANY][ANY]);  // квантование элементов с первым ключом 0
```

- Операция `Lquantize` для именованного массива: печатает ключи и значения конкретного именованного массива в графическом формате с логарифмическим масштабированием значений.

  ```
  lquantize (count);
  ```

  Где count - именованный массив.

  ```
  ключ                  значение
  500                    500                ====
  1000                   1000               ====
  2000                   2000               =====
  4000                   4000               =====
  8000                   8000               ======
  16000                 16000               ======
  32000                 32000               =======
  64000                 64000               =======
  ```

Параметры печати функции **lquantize()** по умолчанию можно переопределить (аналогично функции **print()**).

Пример:

```
/* сортировка именованного массива 'count' в порядке убывания значений и
отображение первых 10 записей 'count' в графическом
формате с учетом логарифмического значения*/
lquantize(count, 10, _BY_VALUE|SORT_TYPE_DESCEND);
```

Для многомерных именованных массивов ключи можно указать в первом аргументе, чтобы указать конкретные элементы:

```
lquantize(var[0][ANY][ANY]);  // квантование элементов с первым ключом 0
```

В следующем примере демонстрируется применение именованного массива:

Пример:

```
# Трассировка всех вызовов alloc и сохранение записи
# Time в именованном массиве 'entry_time'
#
@@uft:$__CPID:*:"/alloc/":entry
{
        entry_time[get_function()]=timestamp();
}
#
# Перед выходом выполняется проверка данных трассировки входа этой функции.
# Если они существуют, удалите время входа из именованного массива 'entry_time', чтобы
# в следующий раз не выполнять никаких действий при выходе, если вход не был зарегистрирован.

@@uft:$__CPID:*:"/alloc/":exit
{
   func =get_function();
   if(exists(entry_time, func) )
   {
             append(time_taken[func],
                   diff_time(timestamp(),entry_time[func],MICROSECONDS));
             delete(entry_time, func);
   }
}
#
# Печать атрибутов sum, min, max, count и avg time, отслеживаемых
# в каждой функции Alloc.
#
```

```
@@syscall:$__CPID:exit:entry
{
  print(time_taken);
  exit();
}
```

Примечание: В этом случае именованные массивы обеспечивают полную поддержку списков без создания нескольких переменных типа List.

Тип данных timestamp:

Переменная типа **probev_timestamp_t** хранит возвращаемое значение функции **timestamp** ProbeVue, которая возвращает системное время во внутреннем формате AIX.

Переменная тип **probev_timestamp_t** может быть передана как параметр в функцию **diff_time**, которая возвращает разницу между значениями системного времени. Этот тип данных можно также хранить в именованном массиве как ключ или как значение.

Хотя компилятор ProbeVue не выполняет проверку типа, когда вы используете тип данных **long** вместо типа данных **probev_timestamp_t** для сохранения значений системного времени, старайтесь избегать этого.

Следующие операции применимы для переменной типа **probev_timestamp_t**:

- Может быть явно инициализирована нулем.

  Примечание: Переменная системного времени глобального класса или класса, локального для нити, инициализируется нулем в начале сеанса ProbeVue.
- Может быть сравнена с нулем. Значения системного времени, возвращенные функцией **timestamp**, всегда больше нуля.
- Может быть сравнена с другой переменной системного времени. Более позднее системное время всегда больше более раннего.
- Может быть передана в качестве параметра в функцию **diff_time**.
- Может быть напечатана с помощью функции **printf** или **trace**.

Тип данных пути к файлу:

Переменную типа `path_t` можно использовать для хранения значения `__file->path` (см. описание встроенной переменной **__file** для администратора тестов ввода-вывода) или `function fd_path()`. Поддерживаются только локальные и глобальные переменные типа `path_t`. Переменная этого типа может также быть ключом или значением в именованном массиве.

Объявление переменной пути к файлу

```
path_t pth; // глобальная переменная типа path_t
auto:pth2 = fd_path(fd); // сохранить в локальной переменной path_t
my_aso[__file->fname] = __file->path; // сохранить в именованном массиве
```

Спецификаторы signed, unsigned, register, static, thread и kernel не поддерживаются для переменных типа `path_t`.

Операции присвоения

Оператор присвоения (=) позволяет присвоить одну переменную `path_t` другой переменной `path_t`. Исходное значение переменной заменяется.

В следующем примере после присвоения переменные p1 и p2 ссылаются на один и тот же путь к файлу:

```
path_t p1, p2;
p1 = fd_path(fd); // fd - допустимое значение дескриптора файла
p2 = p1;
```

Операция сравнения

Для переменных path_t разрешены только операторы равенства (==) и неравенства (!=). Результат оператора равенства истинный (1), если обе переменных представляют один и тот же абсолютный путь к файлу. В противном случае он ложный (0).

Оператор неравенства имеет противоположное поведение. Другие операторы сравнения (>=, >, < и =<) запрещены для переменных типа path_t.

Вывод переменных типа пути к файлу

Переменные типа path_t можно выводить с помощью спецификатора формата "%p" функции printf().

При выводе переменной пути к файлу выполняется затратная по времени операция поиска файла в соответствующей файловой системе. Это необходимо учитывать в сценариях Vue.

Примечание: Для временных файлов, которые могут перестать существовать к моменту вызова printf(), выводится пустая строка вместо пути к файлу.

Именованные массивы, в которых переменные типа path_t играют роль ключей и (или) значений, могут выводиться с помощью функции print().

```
printf("путь к файлу=[%p]\n", __file->path);
my_aso[0] = fd_path(fd); // fd - допустимое значение дескриптора файла
print(my_aso);
```

Ограничения переменных типа пути к файлу

- Нельзя объявлять массив переменных path_t.
- Переменные path_t не могут быть элементами структуры или объединения.
- Запрещены указатели на переменные path_t.
- Запрещено преобразование типа переменной path_t в любой другой тип или преобразование любого другого типа в тип path_t.
- Нельзя использовать арифметические операторы (+, -, *, /, ++, -- и пр.) с переменными типа path_t.

Тип данных MAC-адреса:

Тип данных mac_addr_t используется для хранения MAC-адреса. Тип данных MAC-адреса - это абстрактный тип данных, и его нельзя использовать непосредственно со стандартными унарными или бинарными операторами языка C.

Поддерживаются только локальные и глобальные переменные типа mac_addr_t.

Переменные этого типа можно также хранить в именованном массиве как ключ или как значение.

Язык Vue поддерживает следующие свойства и операции для переменных типа MAC-адрес:

Объявление переменной MAC-адреса

```
mac_addr_t m1;           // глобальная переменная типа MAC-адрес
__auto mac_addr_t m2;                            // автоматическая переменная типа MAC-адрес
m2 = __etherhdr->src_addr;                       // сохранить исходный MAC-адрес в локальной переменной
mac_aso["src_mac_addr"] = __etherhdr->src_addr ;  // сохранить в именованном массиве.
```

Спецификаторы signed, unsigned, register, static, thread и kernel не поддерживаются для переменных типа mac_addr_t.

Операции присвоения

Оператор присвоения (=) позволяет присвоить одну переменную типа mac_addr_t другой переменной типа mac_addr_t. Исходные значения переменной заменяются. Преобразование типа mac_addr_t и другие типы и наоборот запрещено.

В следующем примере переменная mac_addr_t m1 присваивается переменной m2.

```
mac_addr_t m1, m2; // Объявление двух переменных MAC-адресов.
m1 = __etherhdr->src_addr;      // Сохранить исходный MAC-адрес пакета в переменной m1.
m2 = m1 ;                       // Сохранить значение переменной m1 в переменной m2.
```

Операция сравнения

Для переменных mac_addr_t разрешены только операторы равенства (==) и неравенства (!=). Результат оператора равенства истинный (1), если обе переменных содержат один и тот же MAC-адрес, и ложный (0) в противном случае.

Оператор неравенства имеет противоположное поведение. Другие операторы сравнения (>=, >, < и =<) запрещены для переменных типа mac_addr_t.

```
if( m1 == m2)                  // сравнение двух переменных типа mac_addr_t.
printf("MAC-адреса равны"); else printf("MAC-адреса неравны");
```

Вывод переменных типа MAC-адрес

Переменные типа mac_addr_t можно выводить с помощью спецификатора формата "%M" функции printf() сценария Vue. Именованные массивы, в которых переменные типа mac_addr_t играют роль ключей и (или) значений, могут выводиться с помощью функции print().

```
printf(" Source MAC address=[%M]\n", __etherhdr->src_addr);
mac_aso["src_mac_address"] = __etherhdr->src_addr ; // Сохранить исходный MAC-адрес как значение в именованный массив mac_aso.
print(mac_aso);
```

Ограничения переменной типа MAC-адрес

- Нельзя объявить массив переменных mac_addr_t.
- Переменные mac_addr_t не могут быть элементом структуры или объединения.
- Запрещены указатели на переменные типа mac_addr_t.
- Запрещено преобразование переменных типа mac_addr_t в любой другой тип и переменных любых других типов в тип mac_addr_t.
- Нельзя использовать арифметические операторы (+, -, *, /, ++, -- и пр.) с переменными типа mac_addr_t.

Тип данных IP-адреса:

Это абстрактный тип данных, поэтому его нельзя напрямую использовать со стандартными унарными и бинарными операторами языка C. Поддерживаются только локальные и глобальные переменные типа ip_addr_t. Переменные этого типа можно также хранить в именованном массиве как ключ или как значение.

Vue поддерживает следующие свойства и операции для переменных типа IP-адрес:

Объявление переменной IP-адреса

```
ip_addr_t i1;                     // глобальная переменная типа ip_addr_t
__auto ip_addr_t i2;                        // автоматическая переменная типа ip_addr_t
ip_addr_t i2 = __ip4hdr->src_addr; // исходный IP-адрес сохраняется в локальной переменной ip_addr_t.
ip_aso["src_ip_addr"] = __ip4hdr->src_addr; // сохранить в именованном массиве.
```

Спецификаторы signed, unsigned, register, static, thread и kernel не поддерживаются для переменных типа ip_addr_t.

Операции присвоения

Оператор присвоения (=) позволяет присвоить одной переменной типа ip_addr_t значение другой переменной типа ip_addr_t. Кроме того, он позволяет присвоить переменной типа ip_addr_t постоянный IP-адрес или имя хоста. Исходные значения переменной заменяются. Преобразование типа ip_addr_t в другие типы и наоборот запрещено.

В следующем примере переменная i1 типа ip_addr_t присваивается переменной i2.

```
ip_addr_t i1, i2; // Объявление двух переменных IP-адреса.
ip_addr_t i3, i4, i5;               // Объявление трех переменных типа IP-адрес.
i1 = __ip4hdr->src_addr;   // Сохранить исходный IP-адрес пакета в переменной i1.
i2 = i1 ;                  // Сохранить значение переменной i1 в переменной i2.
i3 = "10.10.10.1"; // Переменной i3 присваивается постоянный адрес IPv4.
i4 = "fe80::2c0c:33ff:fe48:f903"; // Переменной i4 присваивается адрес Ipv6.
i5 = "example.com"; // Переменной i5 присваивается имя хоста.
                                  // Копирование содержимого i1 в i2.
```

Операция сравнения

Для переменных ip_addr_t разрешены только операторы равенства (==) и неравенства (!=). Переменные типа ip_addr_t можно сравнивать только друг с другом и со строковой константой (в этом случае IP-адрес или имя хоста указывается в двойных кавычках, например "192.168.1.1" или "example.com").

Результат оператора равенства истинный (1), если обе переменных содержат одинаковые IP-адреса одинакового типа (IPv4 или IPv6). И ложный (0) в противном случае. Оператор неравенства имеет противоположное поведение. Другие операторы сравнения (>=, >, < и =<) запрещены для переменных типа ip_addr_t.

```
if( i1 == i2) // сравнение двух переменных типа ip_addr_t.
                          // Строка IP-адреса
printf("IP-адреса равны");
else printf("IP-адреса неравны");
or
if(i1 == "192.168.1.1") // сравнение переменной типа ip_addr_t со строковой константой.
 printf("IP-адреса равны");
else printf("IP-адреса неравны");
or
if (i1 = "example.com") // сравнение переменной типа ip_addr_t и константы
                             // Строка IP-адреса
printf("IP-адреса равны");
else printf("IP-адреса неравны");
```

Вывод переменных типа IP-адрес

Переменные типа ip_addr_t можно выводить с помощью спецификаторов формата "%I" (вывод в десятичном формате с точками или 16-ричном формате) и "%H" (вывод имени хоста) функции printf() сценария Vue. При выводе имени хоста выполняется продолжительная по времени операция поиска в DNS. Это необходимо учитывать в сценариях VUE.

Примечание: Если пользователь указывает спецификатор формата "%H" для вывода имени хоста для IP-адреса и этого IP-адреса нет в DNS, то вместо имени хоста выводится IP-адрес в десятичном или 16-ричном формате с точками.

Именованные массивы, в которых переменные типа ip_addr_t играют роль ключей и (или) значений, могут выводиться с помощью функции print().

```
printf(" Исходный IP-адрес=[%I]\n", __ip4hdr->src_addr);
ip_aso["src_ip_address"] = __ip4hdr->src_addr ; // Сохранить исходный IP-адрес как значение в именованном массиве
print(ip_aso);
```

Ограничения переменной типа IP-адрес

- Нельзя объявить массив переменных ip_addr_t.
- Запрещены указатели на переменные типа ip_addr_t.
- Запрещено преобразование переменных типа ip_addr_t в любой другой тип и переменных любых других типов в тип ip_addr_t.
- Нельзя использовать арифметические операторы (+, -, *, /, ++, -- и пр.) с переменными типа ip_addr_t.

Тип данных net_info_t:

Переменная `net_info_t` - это структура или составная переменная, содержащая информацию о кортежах (локальные и удаленные IP-адреса и номера портов) из конкретного дескриптора сокета с помощью функции Vue `sockfd_netinfo`.

Доступ к элементам структуры `net_info_t` осуществляется так же, как к элементам любой другой структуры в сценарии Vue. Тип `net_info_t` - это абстрактный тип данных. Переменную такого типа нельзя использовать напрямую в стандартных унарных или бинарных операторах C. Эта переменная представляет собой структуру, содержащую 4-элементный кортеж с информацией. Для обращения к элементам этой переменной используется оператор "." (точка), то есть так же, как для доступа к элементам структур в языке C.

Элементы типа данных `net_info_t`:

```
net_info_t
{
    int local_port;
    int remote_port;
    ip_addr_t local_addr;
    ip_addr_t remote_addr; };
```

Язык Vue поддерживает следующие характеристики и операторы для переменных типа `net_info_t`:

Объявление переменной типа net_info_t

```
net_info_t n1,n2 // n1 - переменная типа net_info_t
sockfd_netinfo(fd, n1);
// fd - дескриптор сокета, а n1 содержит 4-элементный кортеж с
// сетевой информацией из функции Vue sockfd_netinfo.
n2.local_addr = __ip4hdr->src_addr; n2.remote_addr = __ip4hdr->dst_addr; n1.local_port = __tcphdr->src_port;
n1.remote_port = __tcphdr->dst_port;
```

Спецификаторы signed, unsigned, register, static, thread, local, global и kernel не поддерживаются для переменных типа `net_info_t`.

Ограничения переменных типа net_info_t

- Не могут быть элементами структур и объединений.
- Нельзя объявлять указатели на переменные типа `net_info_t`.
- Такие переменные нельзя использовать в именованных массивах.
- Нельзя объявить массив переменных типа `net_info_t`.
- Запрещено преобразование типа переменной `net_info_t` в любой другой тип или преобразование любого другого типа в тип `net_info_t`.
- Нельзя использовать арифметические операторы (+, -, *, /, ++, -- и пр.) с переменными типа `net_info_t`.

Библиотечные функции Vue

В отличие от программ, написанных на языках C или FORTRAN или на внутреннем языке, сценарии, написанные на Vue, не имеют доступа к подпрограммам, предоставленным библиотеками системы AIX или пользовательскими библиотеками. Однако, Vue поддерживает свою собственную внутреннюю библиотеку полезных функций для динамической трассировки программ.

Функции для трассировки

get_function
Возвращает имя функции, включающей текущий тест. Если функция **get_function** вызывается в местах interval, systrace, BEGIN и END, она возвращает пустую строку.

timestamp
Возвращает текущую метку времени.

diff_time
Находит разницу между двумя значениями системного времени в микросекундах или миллисекундах.

Функции захвата трассировки

printf Форматирует и печатает значения переменных и выражений.

trace Печатает данные без форматирования.

stktrace
Форматирует и печатает трассировку стека.

Функции списка

list Создает экземпляр переменной списка.

append
Добавляет новый элемент к списку.

sum, max, min, avg, count
Функции агрегирования, которые можно применять к переменной списка.

Функции библиотеки C

atoi, strstr
Функции стандартной строки.

Функции для поддержки предварительной трассировки

start_tentative, end_tentative
Индикаторы для запуска и завершения предварительно трассировки.

commit_tentative, discard_tentative
Фиксирует или отбрасывает данные предварительной трассировки.

Прочие функции

exit Завершает сценарий Vue.

get_userstring
Читает строку (или данные) из пользовательской памяти.

ptree Печатает дерево процессов текущего процесса.

Строковые функции Vue можно применять только к переменным строкового типа, но не к переменным указателя. Стандартные строковые функции, такие как **strcpy**, **strcat** и так далее, не обязательно относятся к Vue, так как они поддерживаются посредством самого синтаксиса языка.

Компилятор ProbeVue проверяет тип данных параметров, переданных в функции Vue.

В функции **printf** проверяется, совпадает ли каждый аргумент функции **printf** с типом в соответствующем спецификаторе. Количество спецификаторов формата и количество аргументов, переданных функции **printf**, должны совпадать. Кроме того, проверяется, совместим ли фактический тип переданного аргумента с соответствующим спецификатором, указанным в строке формата. Если типы не совместимы, Probevue выводит сообщение об ошибке.

Пример:

```
printf("hello world %s, %d\n", str);
```

вызовет ошибку, поскольку не указано переменной, которая соответствует спецификатору **%d**. Аналогично,

```
Printf("Общее кол-во элементов составляет %d\n", str);
```

вызовет ошибку, поскольку спецификатор **%d** обозначает целочисленный тип, а переменная **str** имеет строковый тип.

Другие функции компонентов

Тем не менее, в выражении

```
printf ("Общее кол-во элементов составляет %lld\n", i);
```

где *i* - переменная типа int, ошибки не возникнет, поскольку переменная *i* совместима с указанным спецификатором формата. Таким образом, типы не проверяются на совпадение, но проверяется их совместимость.

Невозможно поместить функции в раздел предиката оператора Vue.

Предикаты

Невозможно использовать предикаты, когда операторы над точками теста должны выполняться условно. Секция предиката определяется наличием ключевого слова **когда** сразу после секции определения теста. Предикат содержит обычное условное выражение стиля C в круглых скобках.

Существуют некоторые ограничения на использование выражений внутри раздела предиката:

- В предикате не допускаются переменные класса ядра.
- В предикате не допускаются переменные автоматического класса.
- В предикате не допускаются переменные типа с плавающей точкой.
- В предикате не допускаются функции Vue.
- Изменение ориентации в предикатах не поддерживается и не допускается оператор присваивания "=" и его производные, такие как **+=**, **|=** и т. д.
- В предикате не допускаются параметры, начиная с девятого и выше (переменные класса входа **__arg9**, **__arg10** и так далее).

Условное выполнение определенных действий в операторе возможно посредством оператора **if ... else**, который работает подобно аналогичному оператору в языке C. Однако, если весь оператор должен быть выполнен условно, предпочтительнее использовать предикаты, так как ProbeVue предназначен для оптимизации выполнения предикатов.

Примечание: Когда точка теста может быть инициирована для нескольких процессов, использование внутри предиката переменных, локальных для нити, является прекрасным способом уменьшить общее влияние включения теста на производительность. Помещение условных проверок внутри предиката предпочтительно для использования оператора **if**.

Следующий сценарий использует локальные для нити переменные внутри предикатов, чтобы эффективно обнаруживать, когда определенная строка символов записана в указанный файл. Он также является примером использования оператора **if** в блоке действия оператора с предикатом. И имя файла, и строка символов, передаются в качестве параметров в сценарий с помощью позиционных параметров оболочки.

```
/*
 * Имя файла : chkfilewrite.e
 *
 * Обнаружение записи определенного слова в определенный файл
 * принимает 2 аргумента: filename и word
 *
 * предполагается, что имя файла < 128
```

```
 *
 * Формат: probevue chkfilewrite.e \"<имя-файла>\" \"<строка>\"
 *
 * Обратные косые черты выше обязательны для предотвращения
 * отбрасывания оболочкой двойных кавычек.
 */

int open(char *fname, int m, int p);
int write(int fd, char *s, int size);

@@syscall:*:open:entry
{
 __auto String fname[128];

 fname = get_userstring(__arg1, -1);

 if (fname == $1)
  thread:opening = 1;
}

@@syscall:*:open:exit
 when (thread:opening == 1)
{
 thread:fd = __rv;
 thread:opening = 0;

}

@@syscall:*:write:entry
 when (thread:fd == __arg1)
{
 __auto String buf[128];

 if (__arg3 < 128)
  buf = get_userstring(__arg2, __arg3);
 else
  buf = get_userstring(__arg2, 128);

 if (strstr(buf, $2)) {
  printf("%d wrote word to file.\n", __pid);
  exit();
 }
}
```

Для того чтобы запустить эту программу для обнаружения записи кем-либо строки "Error" в файл **foo.log**, можно выполнить следующую команду:

```
probevue chkfilewrite.e \"foo.log\" \"Error\"
```

Примечание: Можно усовершенствовать приведенный выше сценарий, добавив тест **close** для обнаружения закрытия файла, чтобы предотвратить захват слова сценарием, когда исходный файл закрыт и открыт новый файл с использованием того же номера дескриптора файла.

Символьные константы

Язык Vue поддерживает некоторые предопределенные символьные константы, которые обычно используются в программировании для AIX. Эти константы обрабатываются как ключевые слова в Vue. В процессе компиляции константы заменяются своими определениями в системных заголовочных файлах. Символьные константы администраторов тестов описаны в соответствующих разделах. Ниже приведены общие символьные константы.

AF_INET
Задает семейство адресов типа IPv4 для применения данных типа IPV4.

AF_INET6
Задает семейство адресов типа IPv6 для применения данных типа IPV6.

NULL Для того чтобы установить типы указателя в значение NULL или ноль. NULL невозможно использовать, чтобы установить пустую строку для строковой переменной.

Номера ошибок или имена "errno"

Это имена стандартных ошибок, такие как **EPERM**, **EAGAIN**, **ESRCH**, **ENOENT** и т. д., указанные в стандартах POSIX и ANSI и определенные в заголовочном файле **/usr/include/sys/errno.h**.

Следующий сценарий отслеживает, когда системный вызов **bind** не выполнен, и errno равно **EADDRINUSE** (адрес уже используется).

```
/*
 * Файл: bind.e
 */

/*
 * Можно использовать void для параметров, если не планируется
 * доступ к ним в этом сценарию.
 */
int bind(void);

@@syscall:*:bind:exit
 when (__rv == -1)
{
 /*
  * Следующая проверка может быть также перемещена в предикат,
  * хотя это не даст большого выигрыша, так как мы уже находимся в
  * пути ошибки, который должен выполняться редко
  */
 if (__errno == EADDRINUSE)
 /* Эта проверка может быть также перемещена в предикат */
  printf("%d failed with EADDRINUSE for bind() call.\n", __pid);
}
```

Сигнальное имя

Это имена стандартных сигналов, такие как **SIGSEGV**, **SIGHUP**, **SIGILL**, **SIGABRT** и т. д., указанные в стандартах ANSI и определенные в заголовочном файле **/usr/include/sys/signal.h**.

Следующий сценарий показывает, как можно отследить, "кто" уничтожил определенный процесс, отправив сигнал.

```
/*
 * Файл: signal.e
 *
 * Кто отправил SIGKILL в мой процесс ?
 */

/* ИД процессов < 2^32, поэтому использование здесь 'int' вместо pid_t
 * вполне допустимо
 */
int kill(int pid, int signo);

@@syscall:*:kill:entry
 when (__arg1 == $1 && __arg2 == SIGKILL)
{
 /* Отправитель трассировки SIGKILL */
 printf("Stack trace of %s: (PID = %d)\n", __pname, __pid);
 stktrace(PRINT_SYMBOLS|GET_USER_TRACE, -1);
 exit();
}
```

FUNCTION_ENTRY

Определяет, является ли точка теста точкой входа функции. Используется в функции **get_location_point**.

FUNCTION_EXIT

Определяет, является ли точка теста точкой выхода функции. Используется в функции **get_location_point**.

Файлы заголовков

Можно указать несколько файлов заголовка в командной строке, или разделив их запятыми (без пробелов между запятыми и именами файлов), или указав каждый файл отдельно с флагом **-I**. Следующие два примера эквивалентны:

```
probevue -I myheader.i,myheader2.i myscript.e
probevue -I myheader.i -I myheader2.i myscript.e
```

Можно включить заголовочный файл C++ в определения struct/class, что позволит сценарию **probevue** получить доступ к полям этих типов по указателю. Все заголовочные файлы C++ должны быть перечислены между директивами сценария ProbeVue **##C++** и **##Vue** с помощью **#include**. Для использования данной опции в системе должен быть установлен компилятор IBM C++. Другой способ включить заголовочный файл C++ - сначала обработать его с помощью опции **-P** команды **probevue**, а затем включить обработанный файл с помощью опции **-I** команды **probevue**. Команда **probevue** с опцией **-P** создаст выходной файл с таким же именем, как и у заголовочного файла C++, но с суффиксом **.Vue**.

Преимущество использования опции **-I** - отсутствие необходимости в установке компилятора IBM C++.

Можно выполнить следующую команду для предварительной обработки заголовочного файла C++.

```
probevue -P myheader.h
```

Примечание: Для выполнения данной команды необходим компилятор IBM C++.

Приведенная выше команда создаст файл **myheader.Vue**. Этот файл можно перенести на другую систему и использовать для тестирования приложения C++ с помощью опции **-I** команды **probevue**. На целевой системе должно использоваться такое же окружение, как и на исходной системе. Включить заголовочный файл на целевой системе можно с помощью опции **-I** команды **probevue**.

Независимо от способа создания и включения, заголовочный файл C++ должен иметь расширение **.h**. Для включения заголовочного файла IOstream используйте **#include<iostream.h>** вместо **#include<iostream>**.

Для тестирования приложения C++ можно запустить программу **cpp_executable** и сценарий **myscript.e**.

```
probevue -I myheader.Vue -X cpp_executable myscript.e
```

Примечание: Для выполнения данной команды компилятор IBM C++ необязателен.

Поддерживаемые элементы оболочки

Синтаксис языка Vue включает поддержку переменных оболочки, определенных префиксом **$**, таких как экспортированные переменные среды и параметры позиционирования (аргументы сценария).

Переменные оболочки Vue могут появляться в любом месте сценария Vue. Они могут быть частью спецификация теста, использоваться в предикатах или внутри операторов в блоках действия. Однако, в отличие от сценария оболочки, они не расширяются, если используются в строках в двойных кавычках.

Аргументы, передаваемые из командной строки в сценарий, адресуются внутри сценария как**$1**, **$2**, **$3** и т. д. Рассмотрим следующий сценарий Vue:

```
/* Имя программы: myscript.e */
 @@syscall:*:read:entry
  when (__pid == $1)

 {
  int count;
  count++;
 }
```

В следующем примере ИД процесса, в котором работает программа **myprog**, заменяет **$1** в предыдущем сценарии. Предполагается, что программа оболочки **prgrep**, которая печатает ИД процесса, используется для вызова сценария Vue.

```
probevue myscript.e `prgrep myprog`
```

Переменные среды, экспортированные из оболочки, также могут быть вызваны в сценарии с помощью оператора **$**. Рассмотрим следующий сценарий Vue:

```
/* Имя программы:myscript2.e */
 @@syscall:*:read:entry
  when (__pid == $PID)

 {
  int count;
  count++;
 }

 /* трассируемая программа имеет функцию 'foo' */

 @@uft:$PID:*:foo:entry
 {
  printf("Запрос чтения системы был выполнен %d раз\n", count);
 }
```

В следующем примере 3243 заменяет $PID из предыдущего сценария:

```
PID=3423 probevue myscript2.e
```

Если необходимо, чтобы переменная среды была распознана как строка внутри сценария ProbeVue, то значение переменной среды должно быть заключено в двойные кавычки, определяющие ее как строку. Например, следующий сценарий захватывает вывод трассировки, когда определенный файл открывается в системе:

```
/* Имя программы: stringshell.e */
 int open(char *path, int oflag);
 @@syscall:*:open:entry
 {
  String s[40];
  s = get_userstring(__arg1, -1);
  if (s == $FILE_NAME) {
   printf("pid %d (uid %d) opened %s\n",__pid,__uid, s);
   exit();
  }
 }
```

Сценарий ожидает, что $FILE_NAME - это имя экспортированной переменной среды оболочки, значение которой включает в себя двойные кавычки. Следующий сценарий является примером этого:

```
 export FILE_NAME=\"/etc/passwd\"
 probevue stringshell.e
```

Если значение существующей переменной среды, не имеющее двойных кавычек, требуется в сценарии, то необходимо создать новую переменную среды, заключив в двойные кавычки существующую переменную среды. Следующий сценарий является примером этого:

```
 export FILE_NAME=\"$HOME\"
 probevue stringshell.e
```

Vue поддерживает две специальные переменные среды, которые полезны, когда процесс запущен самой командой **probevue** с помощью флага **-X**. Переменная среды **$__CPID** указывает на ИД дочернего процесса, созданного командой **probevue**, а переменная среды **$__CTID** указывает на его ИД нити. Флаг **-X** полезен для тестирования скоротечных процессов, особенно в целях отладки.

Сценарий Vue может быть выполнен непосредственно (подобно сценарию оболочки), для чего необходимо установить первую строку следующего вида:

```
#!/usr/bin/probevue
```

Команда **probevue** может также прочитать сценарий Vue из стандартного ввода, как это делает оболочка. Для этого необходимо пропустить имя файла сценария в командной строке. Это полезно для тестирования коротких сценариев.

Vue не поддерживает специальных параметров оболочки, таких как **$$** и **$@**, которые создаются внутренне оболочкой.

Средства захвата трассировки

ProbeVue поддерживает комплексные средства захвата трассировки. Основное действие захвата трассировки обеспечивается посредством функции **printf**, которую можно вызвать из любого теста в составе блока действия. Версия Vue функции **printf** имеет основную функциональность версии библиотеки C. Второй функцией захвата трассировки является функция **trace**. Функция **trace** принимает одиночную переменную в качестве параметра и копирует ее значение в пригодном для печати шестнадцатеричном формате в буфер трассировки. Эта функция особенно полезна для создания дампа содержимого строки структур. Функция **stktrace** - это другая функция захвата трассировки, которая захватывает трассировку стека трассируемой нити в текущей точке теста.

В дополнение к значениям внутренних переменных сценария, внешние переменные, такие как глобальные переменные ядра, контекстно-зависимые данные (например, параметры тестируемой функции), значения возврата из функции и так далее, также могут быть захвачены и показаны посредством этих функций захвата трассировки.

Программа создания отчетов трассировки всегда показывает данные трассировки в порядке времени их появления, и, таким образом, захваченные из разных CPU данные внутренне сортируются перед выводом.

Предварительная трассировка

Предварительная трассировка позволяет производить разумную фильтрацию данных, уменьшая фактический объем данных трассировки, которые представляются для анализа. Это производит замечательный побочный эффект, предотвращая переполнение буфера, если вы сумеете обеспечить раннюю фиксацию или отбрасывание предварительно собранных данных.

Следующий сценарий является примером использования функций предварительной трассировки для захвата данных трассировки только по необходимости:

```
/*
 * Файл: tentative.e
 *
 * Печать сведений, когда системный вызов записи производится дольше
 * указанного количества микросекунд
 *
 * Формат: probevue tentative.e <ИД-процесса> <микросекунды>
 */
int write(int fd, char *buf, int size);

@@BEGIN
{
 probev_timestamp_t ts1, ts2;
}

@@syscall:$1:write:entry
{
 __auto String buf[256];

 if (__arg3 < 256)
  buf = get_userstring(buf, __arg3);
 else
  buf = get_userstring(buf, 256);

 start_tentative("write");
```

```
 /* печать всех данных, связанных с записью */
 stktrace(PRINT_SYMBOLS|GET_USER_TRACE, -1);

 printf("fd = %d, size = %d\n", __arg1, __arg3);

 /* Печатает 256 байтов из buf, даже хотя размер может быть < 256 */
 trace(buf);

 end_tentative("write");

 /* Получить системное время, когда мы вошли в запись: делать это в конце
  * теста, чтобы уменьшить воздействие теста
  */
 ts1 = timestamp();
}

/* Если тестирование начало в середине записи, то ts1 будет равно нулю,
 * этот случай игнорируется с помощью предиката
 */
@@syscall:$1:write:exit
 when (ts1 != 0)
{
 /* diff_time() может вернуть 64-разрядное значение, но мы используем
  * здесь int, так как мы не ожидаем, что разница будет
  * больше нескольких сотых микросекунды.
  */
 int micros;

 /* Получить системное время, когда мы вышли из записи: делать это в начале
  * теста, чтобы уменьшить воздействие теста
  */
 ts2 = timestamp();

 micros = diff_time(ts1, ts2, MICROSECONDS);

 start_tentative("write");
 printf("Return value from write = %d\n", __rv);
 end_tentative("write");

 if (micros > $2) {
  /* Можно смешать нормальную трассировку с предварительной */
  printf("Time to write = %d, limit =%d micro seconds\n",
   micros, $2);
  commit_tentative("write");
  exit();
 }
 else
  discard_tentative("write");
}
```

Запуск ProbeVue

Динамическая трассировка разрешена только для пользователей с правами доступа администратора.

Идентификация и права доступа

Этим она отличается от средств трассировки в AIX, где проверка прав доступа не настолько строгая. Для выполнения команды **probevue** требуются особые права доступа и этому есть объяснение. Сценарий Vue в определенных условиях может повлиять на производительность системы более серьезно, чем статическое средство трассировки, такое как трассировка системы AIX. Это происходит потому, что точки теста трассировки системы предопределены и доступ к ним ограничен. ProbeVue может в определенных обстоятельствах поддерживать намного больше точек теста, а расположения тестов могут быть определены

почти в любом месте. Кроме того, для выполнения действий трассировки ProbeVue в точке теста может потребоваться намного больше времени, чем для действий трассировки системы в точке теста, поскольку для них возможен только явный сбор данных.

Также следует учесть, что ProbeVue позволяет трассировать процессы и считывать глобальные переменные ядра. И то и другое необходимо контролировать, чтобы избежать нарушения защиты. Сеанс ProbeVue может интенсивно использовать ресурс закрепленной памяти, а ограничение использования ProbeVue, при котором допускаются только пользователи с правами доступа, снижает риск атак с отказом в обслуживании. ProbeVue также позволяет администраторам управлять использованием памяти сеансами ProbeVue с помощью интерфейса SMIT.

Права доступа для динамической трассировки предоставляются разными способами, в зависимости от того, включено ли ролевое управление доступом (RBAC). Дополнительные сведения о включении и выключении RBAC приведены на страницах справки AIX.

Учтите, что в версии Legacy или в режиме с выключенным RBAC идентификация не предусмотрена. Обычные пользователи не могут получить права доступа для ввода команды **probevue** для запуска сеанса динамической трассировки или для ввода команды **probevctrl** для управления ProbeVue. Права доступа к этим функциям могут быть только у администратора. Не выключайте RBAC при использовании ProbeVue, если только не требуется ограничить доступ к этому средству, разрешив доступ только пользователям root.

Режим с включенным RBAC

Получение привилегий в системе RBAC происходит при предоставлении прав доступа. Права доступа представляют собой строку текста, связанную с функциями или командами, которые имеют отношение к защите. Права доступа обеспечивают механизм предоставления прав на выполнение действий, для которых предусмотрены права доступа. Только пользователь с достаточными правами доступа может ввести команду **probevue** и начать сеанс динамической трассировки.

aix.ras.probevue.trace.user.self
Эти права доступа позволяют трассировать соответствующие приложения в пользовательском пространстве. ИД пользователя трассируемого процесса должен совпадать с действительным ИД пользователя, вызывающего команду **probevue**. С помощью этих прав доступа можно включить точки теста, предоставленные администратором тестов uft для ваших процессов. При этом необходимо, чтобы действующий, действительный и сохраненный идентификаторы пользователя трассируемого процесса совпадали. Таким образом, невозможно выполнить трассировку программ setuid, используя только эти права доступа.

aix.ras.probevue.trace.user
С помощью этих прав доступа можно трассировать любое приложение в пользовательском пространстве, включая программы setuid и приложения, запущенные администратором. Будьте внимательны при предоставлении этих прав доступа. С помощью этих прав доступа можно ввести команду **probevue** и включить точки теста, предоставленные администратором тестов uft для любого процесса в системе.

aix.ras.probevue.trace.syscall.self
Эти права доступа позволяют трассировать системные вызовы, создаваемые соответствующими приложениями. Необходимо, чтобы действующий, действительный и сохраненный идентификаторы пользователя процесса, создающего системный вызов, были одинаковыми и совпадали с ИД пользователя, вызвавшего команду **probevue**. С помощью этих прав доступа можно включить точки теста, предоставленные администратором тестов syscall для ваших процессов. Во втором поле спецификации теста должен быть указан ИД процесса для запущенного вами процесса.

aix.ras.probevue.trace.syscall
Эти права доступа позволяют трассировать системные вызовы, создаваемые любым приложением в системе, включая программы setuid и приложения, запущенные администратором. Будьте внимательны при предоставлении этих прав доступа. С помощью этих прав доступа можно ввести команду **probevue** и включить точки теста, предоставленные администратором тестов syscall для

любого процесса. Во втором поле спецификации теста можно указать либо ИД процесса для тестирования определенного процесса, либо * для тестирования всех процессов.

aix.ras.probevue.trace
Эти права доступа позволяют трассировать всю систему. В эти права включены все права доступа, определенные в предыдущих разделах. Можно также обращаться к переменным ядра и считывать их при работе команды **probevue**, а также трассировать события трассировки системы с помощью администратора тестов systrace и трассировать датчики, связанные с процессором, посредством by администратора тестов с интервалами. Будьте внимательны при работе с этими правами доступа.

aix.ras.probevue.manage
Эти права доступа позволяют администрировать ProbeVue. В том числе можно изменять значения различных параметров ProbeVue, запускать и останавливать ProbeVue и показывать сведения о сеансах динамической трассировки любых пользователей при выполнении команды **probevctrl**. Если этих прав доступа нет, то можно использовать команду **probevctrl**, чтобы показать данные сеанса для сеансов динамической трассировки, запущенных вами, или показать текущие значения параметров ProbeVue.

aix.ras.probevue.rase
Эти права доступа позволяют обращаться к набору функций Vue событий RAS, имеющему высокий уровень привилегий, что может привести к созданию записей трассировки системы и трассировки LMT, созданию оперативного дампа и даже к аварийному завершению работы системы. Этими правами доступа следует управлять с особой осторожностью.

aix.ras.probevue
В данном случае предоставляются все права доступа динамической трассировки, что равнозначно объединению всех предыдущих прав доступа.

Администратору (или пользователю root) все эти права доступа присвоены по умолчанию. Другим пользователям права доступа должны присваиваться следующим образом: сначала создается роль с набором прав доступа, а затем эта роль присваивается пользователю. Также пользователю потребуется переключить роли на на роль с требуемыми правами доступа, определенными для динамической трассировки, прежде чем вызвать команду **probevue**. В следующем примере показано, как предоставить права доступа пользователю "joe", чтобы разрешить тесты пользовательского пространства и системных вызовов для процессов, запущенных пользователем "joe".

```
 mkrole authorizations=
  "aix.ras.probevue.trace.user.self,aix.ras.probevue.trace.syscall.self"
  apptrace
chuser roles=apptrace joe
setkst -t roleTR
```

Команда ng:

```
swrole apptrace
```

Примечание: С администратором тестов с интервалами не связаны какие-либо определенные права доступа. Можно разрешить точки теста с интервалами при наличии любых прав доступа из **aix.ras.probevue.trace***.

Права доступа ProbeVue

Права доступа, доступные для ProbeVue, перечислены в следующей таблице. В ней приведено описание каждой привилегии и перечислены связанные с ней права доступа. Привилегии составляют иерархическую структуру, в которой родительская привилегия содержит все права доступа, связанные с привилегиями ее потомков, но может также включать в себя дополнительные привилегии.

Таблица 22. Права доступа ProbeVue

Права доступа	Описание	Права доступа	Связанная команда
PV_PROBEVUE_ TRC_USER_SELF	Позволяет процессу включить точки динамического теста пользовательского пространства в другом процессе с тем же самым действительным ИД пользователя.	aix.ras.probevue.trace.user.self aix.ras.probevue.trace.user aix.ras.probevue.trace aix.ras.probevue	probevue
PV_PROBEVUE_ TRC_USER	Позволяет процессу включить точки динамического теста пользовательского пространства в другом процессе. Включает в себя привилегию PV_PROBEVUE_ TRC_USER_SELF.	aix.ras.probevue.trace.user aix.ras.probevue.trace aix.ras.probevue	probevue
PV_PROBEVUE_ TRC_SYSCALL_SELF	Позволяет процессу включить точки динамического теста системного вызова в другом процессе с тем же самым действительным ИД пользователя.	aix.ras.probevue.trace.syscall.self aix.ras.probevue.trace.syscall aix.ras.probevue.trace aix.ras.probevue	probevue
PV_PROBEVUE_ TRC_SYSCALL	Позволяет процессу включить точки динамического теста пространства системных вызовов в другом процессе. Включает в себя привилегию PV_PROBEVUE_ TRC_SYSCALL_ SELF.	aix.ras.probevue.trace.syscall aix.ras.probevue.trace aix.ras.probevue	probevue
PV_PROBEVUE _TRC_KERNEL	Позволяет процессу обращаться к данным ядра при динамической трассировке.	aix.ras.probevue.trace aix.ras.probevue	probevue
PV_PROBEVUE_ MANAGE	Позволяет процессу управлять ProbeVue.	aix.ras.probevue.manage aix.ras.probevue	probevctrl
PV_PROBEVUE_ RASE	Разрешает использовать ограниченные функции "событий RAS".	aix.ras.probevue.rase aix.ras.probevue	probevue
PV_PROBEVUE_	Равнозначно всем предыдущим привилегиям (PV_PROBEVUE_*) вместе.	aix.ras.probevue	probevue probevctrl

ProbeVue параметры

AIX предоставляет набор параметров, с помощью которых можно настроить ProbeVue или среду ProbeVue. Эти параметры позволяют задать глобальные ограничения на использование ресурсов средой ProbeVue и и ограничения на использование ресурсов каждым пользователем.

Примечание: Администраторы тестов не содержатся в среде ProbeVue, поэтому данные ограничения на них не распространяются.

Все параметры ProbeVue можно изменять с помощью интерфейса SMIT (используйте команду быстрого доступа "smit probevue") или непосредственно командой **probevctrl**. Можно остановить ProbeVue, если активных сеансов динамической трассировки нет, причем для его перезапуска не потребуется перезагрузка. Остановка ProbeVue может быть не выполнена, если были активны какие-либо сеансы, использующие локальные для нити переменные.

В следующей таблице приведен обзор параметров, определенных для сеансов динамической трассировки. В данном описании привилегированный пользователь соответствует администратору или пользователю с правами доступа **aix.ras.probevue.trace**, а непривилегированный пользователь соответствует пользователю, не имеющему таких прав доступа.

Таблица 23. Параметры для сеанса динамической трассировки

Описание, как в SMIT	**Максимальное значение**	**Начальное высокое значение конфигурации**	**Начальное низкое значение конфигурации**	**Минимальное значение**	**Связанная команда**
Максимальный размер закрепленной памяти для среды Probevue	64 ГБ	10 % доступной памяти или максимальное значение в зависимости от того, какая из этих величин меньше.	16 МБ	3 МБ	Максимальный объем закрепленной памяти в МБ, выделенной для структур данных ProbeVue, включая число стеков на CPU и число областей локальных таблиц на CPU, и для всех сеансов динамической трассировки. Сюда не входит память, выделенная администраторами тестов. **Примечание:** Этот параметр можно изменить в любой момент, однако новое значение вступит в силу только после перезапуска ProbeVue.
Стандартный размер буфера трассировки на CPU	256 МБ	128 КБ	8 КБ	4 КБ	Размер буфера по умолчанию в КБ для каждого CPU. Выделяется по два буфера трассировки на CPU для каждого сеанса динамической трассировки ProbeVue: один активный и используемый программой записи или программой на языке Vue при сборе данных трассировки, а другой неактивный и используемый программой чтения или приемником данных трассировки. Например, если на 8-конвейерном процессоре задать размер буфера трассировки на CPU 16 КБ, то общий объем памяти, потребляемой буферами трассировки за сеанс ProbeVue составит 256 КБ. Можно указать другой размер буфера (больше или меньше) при вводе команды **probevue**, если он в рамках ограничений на память сеанса.

Таблица 23. Параметры для сеанса динамической трассировки (продолжение)

Описание, как в SMIT	**Максимальное значение**	**Начальное высокое значение конфигурации**	**Начальное низкое значение конфигурации**	**Минимальное значение**	**Связанная команда**
Максимальный размер закрепленной памяти для сеансов обычных пользователей	64 ГБ	2 МБ	2 МБ	0 МБ	Максимальный объем закрепленной памяти, выделенной для сеанса ProbeVue непривилегированного пользователя, включая буферы трассировки на CPU. Значение 0 эффективно выключает всех непривилегированных пользователей. Для привилегированных пользователей нет никаких ограничений на объем памяти, используемой их сеансами ProbeVue. Но для них продолжает действовать ограничение по максимальному объему закрепленной памяти, выделенной для среды ProbeVue.
Минимальная скорость чтения буфера трассировки для обычного пользователя	5000 мс	100 мс	100 мс	10 мс	Минимальный период в миллисекундах, запрашиваемый непривилегированным пользователем, за который приемник трассировки может проверить данные трассировки. Это значение внутренне округляется до следующего большего числа, кратного 10 мс. Привилегированные пользователи не ограничены этим параметром, но самая высокая скорость чтения, которую можно указать, равна 10 миллисекундам.
Скорость чтения буфера трассировки по умолчанию	5000 мс	100 мс	100 мс	10 мс	Период по умолчанию в миллисекундах, за который приемник трассировки проверяет данные трассировки в буферах трассировки памяти. Можно указать другую скорость чтения (больше или меньше) при запуске команды **probevue**, если это значение будет выше минимального значения скорости чтения буфера.

Таблица 23. Параметры для сеанса динамической трассировки (продолжение)

Описание, как в SMIT	**Максимальное значение**	**Начальное высокое значение конфигурации**	**Начальное низкое значение конфигурации**	**Минимальное значение**	**Связанная команда**
Максимальное число параллельных сеансов для обычного пользователя	8	1	1	0	Число параллельных сеансов ProbeVue, разрешенных для непривилегированного пользователя. Нулевое значение эффективно выключает всех непривилегированных пользователей.
Размер стека вычисления на CPU	256 КБ	20 КБ	12 КБ	8 КБ	Размер стека вычисления на CPU, используемый ProbeVue при вводе сценария Vue. Это значение округляется в большую сторону до следующего числа, кратного 8 КБ. ProbeVue выделяет один стек на CPU для всех сеансов ProbeVue. Память, потребляемая стеками, не включается в ограничения, устанавливаемые для каждого сеанса. **Примечание:** Этот параметр можно изменить в любой момент, однако новое значение вступит в силу только после перекомпоновки и перезагрузки загрузочного образа ядра AIX. Стек ProbeVue необходимо настроить для применения виртуальной памяти размером 96 КБ с целью получения списка текущих каталогов.

Таблица 23. Параметры для сеанса динамической трассировки (продолжение)

Описание, как в SMIT	**Максимальное значение**	**Начальное высокое значение конфигурации**	**Начальное низкое значение конфигурации**	**Минимальное значение**	**Связанная команда**
Размер локальной таблицы на CPU	256 КБ	32 КБ	4 КБ	4 КБ	Размер локальной таблицы на CPU, используемой ProbeVue для хранения переменных автоматического класса памяти, а также для хранения временных переменных. ProbeVue использует половину этой области для автоматических переменных, а оставшуюся половину - для сохранения временных переменных. Это значение всегда округляется в большую сторону до следующего числа, кратного 4 КБ. ProbeVue выделяет одну локальную таблицу и одну временную таблицу на CPU для использования всеми сеансами ProbeVue. Память, занимаемая локальными таблицами, не включается в ограничения, устанавливаемые для каждого сеанса. **Примечание:** Этот параметр можно изменить в любой момент, однако новое значение вступит в силу только после перезапуска ProbeVue.
Минимальный допустимый интервал для теста с интервалами	нет	1		1	Минимальный допустимый временной интервал в миллисекундах для теста с интервалами для глобального пользователя root.

Таблица 23. Параметры для сеанса динамической трассировки (продолжение)

Описание, как в SMIT	Максимальное значение	Начальное высокое значение конфигурации	Начальное низкое значение конфигурации	Минимальное значение	Связанная команда
Число нитей, подлежащих трассировке	нет	32	32	1	максимальное число нитей, поддерживаемых сеансом ProbeVue при наличии локальных переменных нитей. В ходе инициализации сеанса среда ProbeVue выделяет локальные переменные нитей для максимального числа нитей, указанного в этом атрибуте. Если во время выполнения теста число нитей с локальными переменными превысит указанное значение, то сеанс ProbeVue аварийно завершается.
Число страничных ошибок, подлежащих обработке	1024	0	0	0	Число контекстов сбоев страниц для обработки ошибок страниц в пределах всей среды. Контекст сбоев страниц содержит стек и локальную таблицу для сохранения переменных автоматических классов и временных переменных. Контекст сбоев страниц требуется для доступа к данным, выгруженным из памяти. Если свободный контекст сбоев страниц для обработки ошибки страницы отсутствует, то ProbeVue не извлекает выгруженные данные.
Максимальное время выполнения теста для тестов systrace, когда тест запускается в контексте прерывания	нет	0	0	0	Это число ограничивает максимальное время выполнения теста systrace в контексте прерывания (в мс). По умолчанию установлено нулевое значение, то есть тест systrace может выполняться неограниченное время.
Максимальное время выполнения теста для тестов ввода-вывода, когда тест запускается в контексте прерывания	нет	0	0	0	Это число ограничивает максимальное время выполнения теста ввода-вывода в контексте прерывания (в мс). По умолчанию установлено нулевое значение, то есть время не ограничено

Таблица 23. Параметры для сеанса динамической трассировки (продолжение)

Описание, как в SMIT	Максимальное значение	Начальное высокое значение конфигурации	Начальное низкое значение конфигурации	Минимальное значение	Связанная команда
Максимальное время выполнения теста для тестов sysproc, когда тест запускается в контексте прерывания	нет	0	0	0	Это число ограничивает максимальное время выполнения теста sysproc в контексте прерывания (в мс). По умолчанию установлено нулевое значение, то есть время не ограничено.
Максимальное время выполнения теста для тестов сети, когда тест запускается в контексте прерывания	нет	0	0	0	Это число ограничивает максимальное время выполнения теста сети в контексте прерывания (в мс). По умолчанию установлено нулевое значение, то есть время не ограничено.
Максимальный размер сетевого буфера	64 КБ	64 Б	96 Б	96 Б	Это значение - размер заранее выделяемого буфера (в байтах), используемого администратором тестов сети для точек тестирования bpf. Это значение выделяется при включении первого теста bpf и существует в системе до выключения последнего теста bpf. При выключении последнего теста bpf этот буфер освобождается. Этот буфер используется для копирования данных, когда данные пакетов распределены по нескольким буферам пакетов.
Интервал получения асинхронной статистики в миллисекундах	NA	1000 миллисекунд (1 секунда)	1000 миллисекунд (1 секунда)	100 миллисекунд	Интервал получения асинхронной статистики в миллисекундах. Это глобальное значение, которое действует для всех сеансов ProbeVue.
Получать статистику только в асинхронном режиме	NA	No	No	NA	Указывает, что статистику ProbeVue следует собрать в асинхронном режиме даже в том случае, если доступен синхронный режим.
Максимальное время выполнения теста для тестов интервалов, связанных с процессором, когда тест запускается в контексте прерывания.	60 секунд	60 секунд	100 миллисекунд	100 миллисекунд	Это число ограничивает максимальное время выполнения теста тестов интервалов, связанных с процессором, в контексте прерывания (в мс). Значение по умолчанию - 60 сек.

Профилирование сеанса ProbeVue

В среде ProbeVue есть функция профилирования, которую можно включать и выключать для оценки влияния активных тестов на приложение. Эта функция суммирует время, ушедшее на выполнение действий тестов, и выдает отчет по запросу и по завершении сеанса.

Отчет профилирования содержит строку теста и время, затраченное на выполнение действия, связанного с этой строкой. Время выполнения действия теста заносится в список, содержащий общее, минимальное, максимальное и среднее время выполнения действия теста. Данные профилирования также содержат информацию о том, сколько раз измерялось время выполнения действия теста. В случае формирования профайла для нескольких функций в одной строке теста (с помощью регулярного выражения или * вместо имени функции) данные профилирования содержат суммарную информацию для тестов, запущенных для всех таких функций. Информация о времени выполнения функций, которые тестируются отдельно, не включается, включается только информация для всего теста целиком.

Действия теста `BEGIN` и `END` не профилируются этой функцией. Информация профилирования касается только сеанса. Профилирование сеанса probevue можно включить при запуске сеанса с помощью команды **probevue** или **probevctrl**.

Дополнительная информация приведена в описании команд **probevue** и **probevctrl**.

Примеры программ

Пример 1

Следующая каноническая программа "Здравствуй, мир" печатает "Здравствуй, мир" в буфер трассировки и завершает работу:

```
#!/usr/bin/probevue

 /* Hello World in probevue */
 /* Program name: hello.e */

 @@BEGIN
 {
  printf("Hello World\n");
  exit();
 }
```

Пример 2

Следующая программа "Здравствуй, мир" печатает "Здравствуй, мир" при нажатии клавиш Ctrl-C:

```
#!/usr/bin/probevue

 /* Hello World 2 in probevue */
 /* Имя программы: hello2.e */

 @@END
 {
  printf("Hello World\n");
 }
```

Пример 3

Следующая программа показывает использование переменных локальной нити. В этом Vue сценарии подсчитывается число байт, записанных в определенный файл. Подразумевается, что у процессов по одной нити или нити, открывающие файлы, являются теми же нитями, которые выполняют запись в эти файлы. Также подразумевается, что все операции записи выполняются успешно. Сценарий можно остановить в любой момент и получить текущее значение числа записанных байт, введя с терминала Ctrl-C.

```
#!/usr/bin/probevue

 /* Имя программы: countbytes.e */
 int open( char * Path, int OFlag, int mode );
 int write( int fd, char * buf, int sz);
 int done;

 @@syscall:*:open:entry
  when (done != 0 )
 {
  if (get_userstring(__arg1, -1) == "/tmp/foo") {
   thread:trace = 1;
   done = 1;
  }
 }

 @@syscall:*:open:exit
  when (thread:trace)
 {
  thread:fd = __rv;
 }

 @@syscall:*:write:entry
  when (thread:trace && __arg1 == thread:fd)
 {
  bytes += __arg3; /* число байтов - третий аргумент */
 }

 @@END
 {
  printf("Bytes written = %d\n", bytes);
 }
```

Пример 4

В следующей программе предварительной трассировки показано, как сделать так, чтобы трассировка параметров, переданных в системный вызов чтения, выполнялась только в том случае, если при чтении файла **foo.data** возвращается значение 0 байт:

```
#!/usr/bin/probevue
 /* Файл: ttrace.e */
 /* Пример предварительной трассировки */
 /* Сбор данных о параметрах сист. вызова чтения только при сбое чтения */
 int open ( char* Path, int OFlag , int mode );
 int read ( int fd, char * buf, int sz);

 @@syscall:*:open:entry
 {
  filename = get_userstring(__arg1, -1);
  if (filename == "foo.data") {
   thread:open = 1;
   start_tentative("read");
   printf("File foo.data opened\n");
  }
 }

 @@syscall:*:open:exit
  when (thread:open == 1)
 {
    thread:fd = __rv;
    start_tentative("read");
    printf("fd = %d\n", thread:fd);
    thread:open = 0;
 }

 @@syscall:*:read:entry
  when (__arg1 == thread:fd)
```

```
{
 start_tentative("read");
 printf("Read fd = %d, input buffer = 0x%08x, bytes = %d,",
  __arg1, __arg2, __arg3);
 end_tentative("read");
 thread:read = 1;
}

@@syscall:*:read:exit
 when (thread:read == 1)
{
 if (__rv < 0) {
  /* printf ниже, хотя и не предварительный, выполняется только
   * в случае ошибки и объединяется с ранее
   * напечатанными предварительными данными
   */
  printf(" errno = %d\n", __errno);
  commit_tentative("read");
 }
 else
  discard_tentative("read");
 thread:read = 0;
}
```

Возможный вывод, в случае сбоя при чтении из-за неправильного адреса (например: 0x1000), переданного в качестве указателя на буфер входных данных, может иметь следующий вид:

```
#probevue ttrace.e
File foo.data opened
fd = 4
Read fd = 4, input buffer = 0x00001000, bytes = 256, errno = 14
```

Пример 5

Следующий сценарий Vue печатает значения некоторых переменных ядра и сразу завершается. Обратите внимание на функцию **exit** в тесте **@@BEGIN**:

```
/* Файл: kernel.e */
/* Пример обращения к переменным ядра */
/* Структура конфигурации системы из /usr/include/sys/systemcfg.h */
struct system_configuration {
 int architecture; /* архитектура процессора */
 int implementation; /* реализация процессора */
 int version;  /* версия процессора */
 int width;  /* width (32 || 64) */
 int ncpus;  /* 1 = UP, n = n-way MP */
 int cache_attrib; /* атрибуты кэша L1 (битовые флаги) */
    /* бит  0/1 значение  */
    /* -----------------------------------*/
    /* 31  без кэша / кэш имеется */
    /* 30  раздельные I и D / объединенные    */
 int icache_size; /* размер кэша инструкций L1 */
 int dcache_size; /* размер кэша данных L1 */
 int icache_asc;  /* ассоциативность кэша инструкций L1 */
 int dcache_asc;  /* ассоциативность кэша данных L1 */
 int icache_block; /* размер блока кэша инструкций L1 */
 int dcache_block; /* размер блока кэша данных L1 */
 int icache_line; /* размер линии кэша инструкций L1 */
 int dcache_line; /* размер линии кэша данных L1 */
 int L2_cache_size; /* размер кэша L2, 0 = Нет кэша L2 */
 int L2_cache_asc; /* ассоциативность кэша L2 */
 int tlb_attrib;  /* атрибуты TLB (битовые флаги)  */
    /* бит  0/1 значение  */
    /* -----------------------------------*/
    /* 31  без TLB / TLB имеется */
    /* 30  раздельные I и D / объединенные    */
 int itlb_size;  /* записей в TLB инструкций */
```

```
 int dtlb_size;  /* записей в TLB данных */
 int itlb_asc;  /* ассоциативность tlb инструкций */
 int dtlb_asc;  /* ассоциативность tlb данных */
 int resv_size;  /* размер резервирования */
 int priv_lck_cnt; /* счетчик блокировки прокрутки в режиме диспетчера */
 int prob_lck_cnt; /* счетчик блокировки прокрутки в состоянии ошибки */
 int rtc_type;  /* тип RTC */
 int virt_alias;  /* 1 - псевдонимы аппаратного обеспечения поддерживаются */
 int cach_cong;  /* число разрядов страницы для синонима кэша */
 int model_arch;  /* используется системой для определения модели */
 int model_impl;  /* используется системой для определения модели */
 int Xint;  /* используется системой для временного преобразования */
 int Xfrac;  /* используется системой для временного преобразования */
 int kernel;  /* атрибуты ядра       */
    /* бит  0/1 значение     */
    /* ----------------------------------------*/
    /* 31 32-разрядное ядро / 64-разрядное ядро */
                               /* 30   без LPAR      / LPAR               */
                               /* 29   преж. 64-р. ABI / 64-р. Large ABI   */
                               /* 28   без NUMA      / NUMA               */
                               /* 27   UP            / MP                 */
                               /* 26   без доб. DR CPU / доб. DR CPU подд. */
                               /* 25   без DR CPU rm / DR CPU rm поддерж.  */
                               /* 24   без доб. DR MEM / доб. DR MEM подд. */
                               /* 23   без DR MEM rm / DR MEM rm поддерж.  */
                               /* 22   ключи ядра выключены / включены     */
                               /* 21   без восстан.  / восстанов. поддерж. */
                               /* 20   без MLS       / MLS включено        */
 long long physmem; /* байт памяти, доступной для OS */
 int slb_attr;  /* атрибуты SLB         */
    /* бит  0/1 значение     */
    /* ----------------------------------------*/
    /* 31  Под управлением ПО   */
 int slb_size;  /* размер slb (0 = без slb)      */
 int original_ncpus; /* исходное количество CPU      */
 int max_ncpus;  /* макс. число CPU, подд. этим образом AIX  */
 long long maxrealaddr; /* макс. подд. действит. адрес памяти +1    */
 long long original_entitled_capacity;
    /* настроенная процессорная мощность при    */
    /* загрузке, необходимая для утилит LPAR    */
    /* обмена данными между разделами. */
 long long entitled_capacity; /* процессорная мощность           */
 long long dispatch_wheel; /* Период круга диспетчеризации (ед. TB)  */
 int capacity_increment; /* допустимое приращение мощности         */
 int variable_capacity_weight; /* приоритетный вес для распределения*/
     /* мощности при простое */
 int splpar_status; /* Состояние подключения SPLPAR    */
    /* 0x1 => 1=поддержка SPLPAR; 0=нет     */
    /* 0x2 => SPLPAR включен 0=выделенный; */
    /*        1=общий        */
 int smt_status;  /* Состояние подключения SMT                */
    /*    0x1 = поддержка SMT  0=нет/1=да       */
    /*    0x2 = SMT включен  0=нет/1=да         */
    /*    0x4 = нити SMT связаны (true) 0=нет/1=да*/
 int smt_threads; /* Число нитей SMT на физический CPU        */
        int vmx_version;        /* версия VMX, определенная RPA, 0=нет/выкл.*/
 long long sys_lmbsize; /* Размер LMB в этой системе.     */
 int num_xcpus;  /* Число исключающих CPU на линии   */
 signed char errchecklevel;/* Уровень проверки ошибок ядра */
 char pad[3];  /* отступ до границы слова */
        int dfp_version;        /* версия DFP, определенная RPA, 0=нет/выкл.*/
    /* если задан MSbit, то имитируется DFP     */
};

__kernel struct system_configuration _system_configuration;

@@BEGIN
```

```
{
 String s[40];
 int j;
 __kernel int max_sdl; /* Атомарный уровень расщепления системы RAD */
 __kernel long lbolt; /* Тактов после загрузки */

 printf("Число подключенных CPU\t\t= %d\n", _system_configuration.ncpus);

 /* Печать состояния SMT */
 printf("Состояние SMT\t\t\t=");
 if (_system_configuration.smt_status == 0)
  printf(" Нет");
 else {
  if (_system_configuration.smt_status & 0x01)
   printf(" Поддержка");
  if (_system_configuration.smt_status & 0x02)
   printf(" Включен");
  if (_system_configuration.smt_status & 0x04)
   printf(" BoundThreads");
 }
 printf("\n");

 /* Печать уровня проверки ошибок */
 if (_system_configuration.errchecklevel == 1)
  s = "Минимальный";
 else if (_system_configuration.errchecklevel == 3)
  s = "Обычный";
 else if (_system_configuration.errchecklevel == 7)
  s = "Подробный";
 else if (_system_configuration.errchecklevel == 9)
  s = "Максимальный";
 printf("Уровень проверки ошибок\t\t= %s\n",s);

 printf("Атомарный уров. подр. сист. RAD\t= %d\n", max_sdl);

 /* Long в ядре соотв. 64 разр., поэтому используйте %lld ниже */
 printf("Число тактов после загрузки\t= %lld\n", lbolt);

 exit();
}
```

Ниже приведен возможный вывод, полученный в результате выполнения описанного сценария в выделенном разделе Power 5 с атрибутами ядра по умолчанию:

```
# probevue kernel.e
Число подключенных CPU          = 4
Состояние SMT                   = Поддержка Включен BoundThreads
Уровень проверки ошибок         = Обычный
Атомарный уров. подр. сист. RAD = 2
Число тактов после загрузки     = 34855934
```

Администраторы тестов

Администраторы тестов не входят в состав базовой среды ProbeVue. Администраторы тестов включают точки теста, которые могут использоваться ProbeVue для динамической трассировки.

Администраторы тестов обычно поддерживают набор тестовых точек, которые относятся к некоторому общему домену и совместно используют общую функцию или атрибут, отличающий их от других тестовых точек. Тестовые точки эффективно используются в тех точках, где поток управления существенно изменяется, в точках изменения состояния или в других значимых точках. Администраторы тестов действуют осмотрительно и выбирают только те точки, которые расположены в местах безопасного управления.

Администраторы тестов могут выбрать вариант, при котором они будут определять собственные точные правила для спецификаций тестов в рамках общего стиля, которому подчиняются все спецификации тестов.

ProbeVue поддерживает следующие администраторы тестов:

Администратор тестов системных вызовов: Администратор тестов syscall поддерживает тесты на входе и выходе точно определенных и зарегистрированных базовых системных вызовов AIX. У этих системных вызовов одинаковый интерфейс в точке входа **libc.a** (или библиотеки C) и в точке входа ядра. Либо системный вызов должен быть удаленным (библиотека C просто импортирует символ из ядра и затем экспортирует его без кода в библиотеку), либо должен существовать упрощенный код для интерфейса внутри библиотеки.

Администратор тестов syscall принимает 4-кортежную спецификацию тестов в одном из следующих форматов:
- `syscall:*:<имя-системного-вызова>:entry`
- `syscall:*:<имя-системного-вызова>:exit`

где значение поля *имя-системного-вызова* должно быть заменено на имя действительного системного вызова. Это значит, что тест должен быть помещен на входе и выходе системных вызовов. Если второму полю присвоить значение *, то тест будет активироваться для всех процессов.

Примечание: Для включения тестов системных вызовов требуются другие права доступа. Для тестирования всех процессов в системе требуются привилегии более высокого уровня, чем для тестирования собственных процессов.

Дополнительно администратор тестов syscall также принимает 4-кортежную спецификацию тестов в одном из следующих форматов:
- `syscall:<ИД-процесса>:<имя-системного-вызова>:entry`
- `syscall:<ИД-процесса>:<имя-системного-вызова>:exit`

где ИД-процесса можно указать в качестве второго поля спецификации теста для поддержки тестирования определенных процессов.

В качестве имен системных вызовов администратор очередей принимает имена интерфейсов *libc.a*, а не внутренние имена системных вызовов ядра. Например, *libc.a* экспортирует функцию **read**, но фактическое имя системного вызова или точка входа ядра - **kread**. Администратор тестов syscall выполнит внутреннее преобразование интерфейса *libc* в соответствующую точку входа ядра и включит тест на входе в функцию ядра **kread**. Поэтому, если несколько интерфейсов библиотеки C вызовут функцию **kread**, тестовая точка также активируется и для этих интерфейсов. Обычно это не приводит к ошибке, поскольку для большинства системных вызовов, поддерживаемых администратором тестов syscall, существует однозначное преобразование между интерфейсом *libc* и функцией ядра.

Для каждого теста syscall существует равнозначная тестовая точка в библиотечном коде, предоставленном администратором тестов uft. Администратор тестов uft не поддерживает все библиотечные интерфейсы (если только это не удаленный интерфейс и в библиотеке отсутствует код для его вызова или ссылки на него), включая не поддерживаемые администратором тестов syscall. Однако у администратора тестов syscall имеются следующие преимущества:
- Администратор тестов syscall может тестировать любой процесс в системе, указывая звездочку во втором поле.
- Администратор тестов syscall эффективнее администратора тестов uft, так как ему не требуется переключаться из пользовательского режима в режим ядра и обратно для выполнения действий теста.

Дополнительные сведения с полным списком системных вызовов, поддерживаемых администратором тестов syscall, приведены в разделе ProbeVue.

Администратор тестов UFT:

Администратор тестов uft (трассировки пользовательских функций) поддерживает тестирование функций пользовательского пространства, видимых в таблице имен XCOFF процесса. Администратор тестов uft

поддерживает только тестовые точки, являющиеся точками входа или выхода и функций с исходным кодом на языках C или FORTRAN, причем в таблице имен могут содержаться идентификаторы с исходным текстом на языках, отличающихся от C или FORTRAN.

С точки зрения пользователя трассировка приложений на Java™ схожа с существующим механизмом трассировки. JVM при этом выполняет большинство задач от имени Probevue.

Администратор тестов uft принимает 5-кортежную спецификацию тестов в следующем формате:

```
uft:<ИД-процесса>:*:<имя-функции>:<entry|exit>
```

Примечание: Для администратора тестов uft необходимо указать ИД процесса, который требуется трассировать, и полное имя функции, на точке входа или выхода которой требуется поместить тест.

Кроме того, для администратора тестов uft необходимо, чтобы в третьем поле был помещен символ *, указывающий, что имя функции следует искать во всех модулях, загружаемых в адресное пространство процесса, включая главный исполняемый файл и общие модули. Этим подразумевается, что в случае, если программа содержит более одной функции C с этим именем (например, функции со статическим классом, содержащиеся в различных объектных модулях), то тесты будут применяться к точке входа каждой из этих функций.

Для тестирования функции в модуле необходимо указать название модуля в третьем поле. Синтаксис:

```
# Функция foo в любом модуле
@@uft:<pid>:*:foo:entry
# Функция foo в любом модуле любого архива с именем libc.a
@@uft:<pid>:libc.a:foo:entry
# Функция foo в модуле shr.o любого архива с именем libc.a
@@uft:<pid>:libc.a(shr.o):foo:entry
```

Имя функции в четвертой записи можно указать в виде расширенного регулярного выражения (ERE). Выражение ERE необходимо заключить в **"/** и **/"**, например **"/<ERE>/"**.

Когда имя функции указано в виде ERE, тестируются все функции в указанном модуле, имя которых соответствует выражению ERE.

```
 /* Тестируются точки входа всех функций libc.a, начинающиеся с "malloc" */
@@uft:$__CPID:libc.a: "/^malloc.*/":entry
/* Тестируются точки выхода всех функций в a.out */
@@uft:$__CPID:a.out:"/.*/":exit
```

При тестировании функций, имя которых соответствует регулярному выражению, нельзя получить доступ к параметрам функций. Для вывода функции и ее аргументов можно использовать функцию **print_args**. Тип аргументов при выводе будет взят из таблицы обратной трассировки.

При тестировании точки выхода функций, имя которых соответствует регулярному выражению, нельзя получить доступ к возвращаемому значению.

Probevue поддерживает включение тестов в нескольких процессах одновременно. Однако потребуются права доступа даже для тестирования собственных процессов.

Probevue вводит ограничение, делающее невозможной отладку процессов с тестами пользовательского пространства с помощью API на основе **ptrace** или **procfs**.

Как сказано выше, администратор тестов uft поддерживает тесты в общих модулях, таких как модули общих библиотек. В следующем сценарии показан пример трассировки мьютекс-операции путем включения тестов в функциях блокировки и разблокировки мьютекса библиотеки нити.

```
/* pthreadlocks.e */
/* Трассировка мьютекс-операции pthread для многонитевого процесса */
/* Следующие определения взяты из /usr/include/sys/types.h */
```

```
typedef long long pid_t;
typedef long long thread_t;

typedef struct {
 int __pt_mutexattr_status;
 int __pt_mutexattr_pshared;
 int __pt_mutexattr_type;
} pthread_mutexattr_t;

typedef struct __thrq_elt thrq_elt_t;

struct __thrq_elt {
 thrq_elt_t *__thrq_next;
 thrq_elt_t *__thrq_prev;
};

typedef volatile unsigned char _simplelock_t;

typedef struct __lwp_mutex {
 char  __wanted;
 _simplelock_t __lock;
} lwp_mutex_t;

typedef struct {
 lwp_mutex_t  __m_lmutex;
 lwp_mutex_t  __m_sync_lock;
 int   __m_type;
 thrq_elt_t  __m_sleepq;
 int   __filler[2];
} mutex_t;

typedef struct {
 mutex_t   __pt_mutex_mutex;
 pid_t   __pt_mutex_pid;
 thread_t  __pt_mutex_owner;
 int   __pt_mutex_depth;
 pthread_mutexattr_t __pt_mutex_attr;
} pthread_mutex_t;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

@@uft:$__CPID:*:pthread_mutex_lock:entry
{
 printf("thread %d: mutex 0x%08x locked\n", __tid, __arg1);
}

@@uft:$__CPID:*:pthread_mutex_unlock:entry
{
 printf("thread %d: mutex 0x%08x unlocked\n", __tid, __arg1);
}
```

- Пользователь должен связать типы данных языка Fortran с типами данных ProbeVue и использовать данную привязку в сценарии. Ниже приведена таблица соответствия между базовыми типами данных языка Fortran и типами данных ProbeVue.

Таблица 24. Соответствие типов данных языка Fortran и ProveVue

Тип данных языка Fortran	Тип данных ProbeVue
INTEGER * 2	short
INTEGER * 4	int/long
INTEGER * 8	long long
REAL	float
DOUBLE PRECISION	double

COMPLEX	Соответствующий тип данных отсутствует. Данный тип данных необходимо преобразовать в следующую структуру:
	`typedef struct complex {`
	`float a;`
	`float b;`
	`} COMPLEX;`

LOGICAL	int (стандарт языка Fortran требует, чтобы логические переменные занимали в памяти столько же места, сколько и переменные INTEGER/REAL)
CHARACTER	char
BYTE	signed char

- В языке Fortran скалярные аргументы внутренних процедур передаются по значению, остальные - по ссылке. Доступ к аргументам, передающимся по ссылке, должен осуществляться с помощью copy_userdata(). Дополнительная информация о связях аргументов в языке Fortran приведена в разделе Связи аргументов.
- Названия процедур в Fortran не учитывают регистр символов. Но в сценарии ProbeVue имена данных процедур необходимо указывать в нижнем регистре.

  Следующий пример демонстрирует связь между типами данных Fortran и ProbeVue:

```
/* cmp_calc.e */
/* Трассировка процедур Fortran
cmp_calc(COMPLEX, INTEGER) и
cmplxd(void) */

typedef struct complex{
        float a;
        float b;
        } COMPLEX;

typedef int INTEGER;

/* в качестве аргументов используются указатели, передаваемые по ссылке */
void cmp_calc(COMPLEX *, INTEGER *);
void cmplxd();

@@uft:$__CPID:*:cmplxd:entry
{
printf("На входе cmplxd \n");
}

@@uft:$__CPID:*:cmp_calc:entry
{
COMPLEX c;
int     i;
copy_userdata(__arg1, c);
copy_userdata(__arg2, i);
printf("%10.7f+j%9.7f  %d \n", c.a,c.b,i);
}
```

- Fortran хранит массивы с помощью развертывания по столбцам (column-major), а ProbeVue - с помощью развертывания по строкам (row-major). Ниже представлен пример сценария для извлечения элементов массива.

```
/* array.e*/
/* Сценарий ProbeVue для тестирования программы на Fortran array.f */

void displayarray(int **, int, int);
```

```
@@uft:$__CPID:*:displayarray:entry
{
int a[5][4];  /* размеры строк и столбцов меняются местами */
copy_userdata(__arg1, a);
/* выводится первая строка */
printf("%d %d %d \n", a[0][0], a[1][0], a[2][0]);
/* для вывода второй строки */
printf("%d %d %d\n", a[0][1], a[1][1], a[2][1]);
}

/* программа Fortran array.f */

PROGRAM ARRAY_PGM
IMPLICIT NONE
INTEGER, DIMENSION(1:4,1:5) :: Array
INTEGER :: RowSize, ColumnSize
CALL ReadArray(Array, RowSize, ColumnSize)
CALL DisplayArray(Array, RowSize, ColumnSize)
CONTAINS
SUBROUTINE ReadArray(Array, Rows, Columns)
IMPLICIT NONE
INTEGER, DIMENSION(1:,1:), INTENT(OUT) :: Array
INTEGER, INTENT(OUT) :: Rows, Columns
INTEGER :: i, j
READ(*,*) Rows, Columns
DO i = 1, Rows
READ(*,*) (Array(i,j), j=1, Columns)
END DO
END SUBROUTINE ReadArray
SUBROUTINE DisplayArray(Array, Rows, Columns)
IMPLICIT NONE
INTEGER, DIMENSION(1:,1:), INTENT(IN) :: Array
INTEGER, INTENT(IN) :: Rows, Columns
INTEGER :: i, j
DO i = 1, Rows
WRITE(*,*) (Array(i,j), j=1, Columns )
END DO
END SUBROUTINE DisplayArray
END PROGRAM ARRAY_PGM
```

- Встроенные функции нельзя тестировать с помощью ProbeVue . Тестировать можно все процедуры FORTRAN, перечисленные в таблице имен XCOFF. ProbeVue использует данную таблицу для получения сведений о том, где находятся указанные процедуры. Пользователь должен предоставить прототип процедуры. ProbeVue пытается получить доступ к аргументам на основе указанного прототипа. В случае использования компилятора, изменяющего имена процедур, необходимо указать измененное имя. Необходимо убедиться в том, что прототип с функциями FORTRAN соответствующим образом преобразовывается в прототип с функциями на C. См. соглашения по связыванию для передаваемых аргументов и возвращаемых значений в разделе Передача данных из одного языка в другой. Это продемонстрировано в следующем примере:

```
/* Программа на Fortran ext_op.f */
/* Оператор "*" перегружен для умножения рациональных чисел */
MODULE rational_arithmetic
IMPLICIT NONE
        TYPE RATNUM
                INTEGER :: num, den
        END TYPE RATNUM
        INTERFACE OPERATOR (*)
                MODULE PROCEDURE rat_rat, int_rat, rat_int
        END INTERFACE
        CONTAINS
        FUNCTION rat_rat(l,r) ! rat * rat
                TYPE(RATNUM), INTENT(IN) :: l,r
                TYPE(RATNUM) :: val,rat_rat
                val.num=l.num*r.num
                val.den=l.den*r.den
                rat_rat=val
        END FUNCTION rat_rat
```

```
        FUNCTION int_rat(l,r) ! int * rat
                INTEGER, INTENT(IN) :: l
                TYPE(RATNUM), INTENT(IN) :: r
                TYPE(RATNUM) :: val,int_rat
                val.num=l*r.num
                val.den=r.den
                int_rat=val
        END FUNCTION int_rat
        FUNCTION rat_int(l,r) ! rat * int
                TYPE(RATNUM), INTENT(IN) :: l
                INTEGER, INTENT(IN) :: r
                TYPE(RATNUM) :: val,rat_int
                val.num=l.num*r
                val.den=l.den
                rat_int=val
        END FUNCTION rat_int
END MODULE rational_arithmetic
PROGRAM Main1
Use rational_arithmetic
IMPLICIT NONE
 TYPE(RATNUM) :: l,r,l1
 l.num=10
        l.den=11
        r.num=3
        r.den=4
        L1=l*r
END PROGRAM Main1

/* ext_op.e */
/* Сценарий ProbeVue, вызываемый при использовании оператора "*"
    для умножения рациональных чисел в ext_op.f */

struct rat
{
        int num;
        int den;
};
struct rat rat;
void __rational_arithmetic_NMOD_rat_rat(struct rat*,
 struct rat*,struct rat*);
/* Обратите внимание, что приведено измененное имя функции. */
/* Кроме того, возвращаемая структура будет отправлена в буфер, адрес которого задан в первом аргументе. */
/* Первый явный параметр находится во втором аргументе. */
@@BEGIN
{
        struct rat* rat3;
}
@@uft:$__CPID:*:__rational_arithmetic_NMOD_rat_rat:entry
{
 struct rat rat1,rat2;
        copy_userdata((struct rat *)__arg2,rat1);
        copy_userdata((struct rat *)__arg3,rat2);
        rat3=__arg1;
 /* Адрес буфера, в котором будет храниться возвращенная структура, сохраняется при входе функции */
        printf("Argument Passed rat_rat = %d:%d,%d:%d\n",rat1.num,rat1.den,rat2.num,rat2.den);
}
@@uft:$__CPID:*:__rational_arithmetic_NMOD_rat_rat:exit
{
        struct rat rrat;
        copy_userdata((struct rat *)rat3,rrat);
        /* Для обращения к структуре используется сохраненный адрес буфера */
        printf("Return from rat_rat = %d:%d\n",rrat.num,rrat.den);
        exit();
}
```

- ProbeVue не поддерживает прямое включения в сценарий заголовочных файлов Fortran. Связь между типами данных Fortran и ProbeVue можно задать в заголовочном файле ProbeVue. Этот файл можно указать с помощью опции "-I".

Информация, связанная с данной:

Администратор тестов приложений Java

Администратор тестов приложений на c++: Администратор тестов C++ поддерживает тестирование приложений на C++ аналогично администратору тестов C. Поддерживаются тесты в стиле uft (точки входа/выхода) для любых функций C++, в том числе для функций-членов, перегруженных функций, операторов и шаблонных функций. Для тестирования на входе/выходе функций C++ необходимо использовать администратор тестов **@@uftxlc++**.

Все кортежи в спецификации тестовой строки **@@uftxlc++** используются также, как и тестовая строка **@@uft** , за исключением названия функций. Поскольку в C++ можно перегружать функции, то для однозначной идентификации функций необходимо включить типы аргументов в тестовую строку.

Пример:

```
@@uftxlc++:12345:*:"foobar(int, char *)":entry
```

Примечание: Возвращаемый тип данных отсутствует в указанной тестовой строке, поскольку он не нужен для алгоритма изменения имени обычных функций. В случае шаблонной функции необходимо явно указать тестируемый экземпляр шаблона, а также возвращаемый тип.

```
@@uftxlc++:12345:*:void foobar<int>(int, char *):entry
```

Примечание: Название функции в тестовой строке должно заключаться в двойные кавычки, иначе команда probevue выдаст ошибку. Двойные кавычки необходимы из-за использования двоеточия ":" и запятой",". Запятая используется для разделения нескольких тестов на одной строке. Без использования кавычек запятая будет иметь больший приоритет. Это может привести к непонятным для пользователя сообщениям.

При тестировании функции-члена или функции, определенной в пространстве имен, в тестовой строке необходимо указать полное имя функции. Для устранения неоднозначности между разделителем кортежей двоеточием (:) и оператором разрешения области действия двойным двоеточием (::) необходимо заключить полный кортеж имени функции в двойные кавычки.

```
@@uftxlc++:12345:*:"Foo::bar(int)":entry
```

Ограничения:

1. Доступ к полям данных, унаследованным от виртуального базового класса, не поддерживается.
2. Классы шаблона не поддерживаются и не должны быть включены в заголовок C++.
3. Указатели на элементы не поддерживаются.
4. Для тестирования класса с помощью определения класса создается экземпляр объекта класса в файле заголовка или как глобальный объект, или в фиктивной функции.

Пример:

Приложение на c++

```
#include "header.cc"
main()
{
int             i = 10;
incr_num(i);
float           a = 3.14;
incr_num(a);
char            ch = 'A';
incr_num(ch);
double          d = 1.11;
incr_num(d);
}
```

Содержание файла "header.cc"

```
# cat header.cc
#include <iostream.h>
template <class T>
T incr_num( T a)
```

```
{
return (++a);
}
int dummy()
{
int  i=10,j=20;
incr_num(i);
float a=3.14;
incr_num(a);
char  ch ='A',dh='Z';
incr_num(ch);
double d=1.1,e=1.11;
incr_num(d);
return 0;
}
```

Содержание файла vue_cpp.e

```
##C++
#include "header.cc"
##Vue
@@uftxlc++:$__CPID:*:"incr_num<int>(int)":entry
{
printf("Hello1_%d\n",__arg1 );
}
@@uftxlc++:$__CPID:*:"incr_num < float > (float)" :entry
{
printf("Hello2_%f\n",__arg1 );
}
@@uftxlc++:$__CPID:*:"incr_num < char > ( char )":entry
{
printf("Hello3_%c\n",__arg1 );
}
@@uftxlc++:$__CPID:*:"incr_num < double > ( double )":entry
{
printf("Hello4_%lf\n",__arg1 );
exit();
}
```

Выполнение:

```
/usr/vacpp/bin/xlC  app.c++
#  probevue  -X ./a.out  vue_cpp.e
Hello1_10
Hello2_3.140000
Hello3_A
Hello4_1.110000
```

Прототип функции в четвертой записи можно указать в виде расширенного регулярного выражения (ERE). Выражение ERE необходимо заключить в **‘”/’** и **‘/”’** , например **"/<ERE>/"**. Когда прототип функции указан в виде ERE, тестируются все функции в указанном модуле, прототип которых соответствует выражению ERE.

```
/* Тестируются точки входа всех функций C++ в файле a.out */
@@uftxlc++:$__CPID:a.out:"/.*/":entry
/* Тестируются точки выхода всех функций C++, содержащие слово 'foo' */
@@uftxlc++:$__CPID:*:"/foo/":exit
```

При тестировании функций, имя которых соответствует регулярному выражению, нельзя получить доступ к параметрам функций. Для вывода функции и ее аргументов можно использовать функцию **print_args()**. Тип аргументов при выводе будет взят из таблицы обратной трассировки.

При тестировании точки выхода функций, имя которых соответствует регулярному выражению, нельзя получить доступ к возвращаемому значению.

Администратор тестов приложений Java: Администратор тестов Java (JPM) поддерживает тестирование приложений на Java аналогично администратору тестов для C и C++. Один сценарий Vue должен быть способен одновременно трассировать многочисленные приложения на Java, используя ИД различных процессов JVM. Один и тот же сценарий можно использовать для тестирования системных вызовов или приложений на C/C++ и Java. Также можно использовать различные администраторы тестов.

Подобно администратору тестов uft (трассировка пользовательских функций), администратор тестов java использует 5-кортежную спецификацию теста в следующем формате:

```
uftjava :< ИД-процесса> :*:< _полное-имя-функции >: entry
```

Где второй кортеж - это ИД процесса JVM, в которой выполняется трассируемое приложение Java.

Третье поле зарезервировано для использования в будущем.

В четвертом поле необходимо указать полное имя функции java.

Например: Mypackage.Myclass.Mymethod.

Ограничения:

- Тестировать можно только методы на java. Методы типа Native (вызовы общих библиотек), а также зашифрованные данные не поддаются тестированию.
- Поддерживается только тестирование на точке входа.
- Поддерживаются только JVM 1.5 и выше, поддерживающие интерфейс JVMTI.
- Два сеанса Probevue не могу одновременно трассировать одно и то же приложение Java с помощью @@uftjava.
- Переопределенные и перегруженные методы не поддерживаются.
- Не поддерживается трассировка и доступ к внешним переменным с именами, эквивалентными названиям ключевых слов или встроенных функций Probevue. Для этого необходимо переименовать внешние переменные Java.
- Доступ к массивам java приложений не поддерживается в данном выпуске.
- Доступ к массивам java приложений не поддерживается в данном выпуске.
- Встроенная функция get_function () для java не поддерживается в данном выпуске.

Примечание: В случае трассировки нестатического метода нумерация аргументов начинается с __arg2 (как в нестатических методах C++). __arg1 используется в качестве указателя на самого себя.

Доступ к данным. Блоки действий тестов java могут получать доступ к следующим данным.

- Блок действий получает доступ к глобальным, локальным и переменным ядра сценария.
- Блок действий получает доступ к аргументам метода или базовым типам.
- Блок действий получает доступ ко встроенным переменным.
- Блок действий получает доступ к статическим (члены класса) переменным приложений Java с помощью полного имени.

  ```
  x = some_package.app.class.var_x;    //Доступ к статическим переменным/членам класса.
  ```
- Поддерживается доступ к базовым типам. Данные типы должны быть явно преобразованы в соответствующие типы на языке Vue без потери значения. Но фактическое использование памяти в данном случае может отличаться от использования памяти базовыми типами Java.

В следующей таблице перечислены функции, поддерживаемые в контексте администратора тестов Java:

Таблица 25. Функции, поддерживаемые администратором тестов Java

Функция	Описание
stktrace()	Трассировка стека приложения (нити) Java.
copy_userdata()	Копирование данных из приложения java в переменные сценария.
get_probe()	Возвращает тестовую строку.
get_stktrace	Возвращает трассировку стека текущей среды выполнения.
get_location_point()	Возвращает имя текущего расположения теста.
get_userstring()	Копирование строковых данных из приложения java.
exit()	Завершение сеанса трассировки probevue.

Изменения в синтаксисе команды Probevue:

Таблица 26. Изменение команды probevue

Команда	Описание
параметр -X	Данный параметр можно использовать (вместе с параметром -A) для запуска приложения Java. В текущем выпуске пользователь должен дополнительно вручную передать строку agentlib:probevuejava.

Например:

```
probevue -X /usr/java5/bin/java -A  -agentlib:probevuejava myjavaapp  myscript.e
```

При использовании 64-разрядной JVM необходимо использовать "agentlib:probevuejava64":

```
probevue -X /usr/java5_64/bin/java -A  -agentlib:probevuejava64 myjavaapp  myscript.e
Где myjavaapp - название класса приложения myjavaapp
```

Пример класса ExtendedClass.java:

```
class BaseClass
{
        static int i=10;

        public static void test(int x)
        {
                i += x;
        }
}

public class ExtendedClass extends BaseClass
{
        public static void test(int x, String msg)
        {
                i += x;
                System.out.print("Java: " + msg + "\n\n");
                BaseClass.test(x);
        }

        public static void main(String[] args)
        {
                BaseClass.test(5);
                ExtendedClass.test(10, "hello");
        }
}
```

Пример сценария test.e для приведенного выше приложения Java:

```
@@uftjava:$__CPID:*:"BaseClass.test":entry
{
        printf("BaseClass.i: %d\n", BaseClass.i);
        printf("BaseClass.test: %d\n", __arg1);
        stktrace(0, -1);
        printf("\n");
}
```

```
@@uftjava:$__CPID:*:"ExtendedClass.test":entry
{
        printf("BaseClass.i: %d\n", BaseClass.i);
        printf("ExtendedClass.test: %d, %s\n", __arg1, __arg2);
        stktrace(0, -1);
        printf("\n");
}
```

Пример сеанса ProbeVue с приведенным выше сценарием:

```
# probevue -X /usr/java5/jre/bin/java \
-A "-agentlib:probevuejava ExtendedClass" test.e
Java: hello

BaseClass.i: 10
BaseClass.test: 5
BaseClass.test()+0
ExtendedClass.main()+1

BaseClass.i: 15
ExtendedClass.test: 10, hello
ExtendedClass.test()+0
ExtendedClass.main()+8

BaseClass.i: 25
BaseClass.test: 10
BaseClass.test()+0
ExtendedClass.test()+39
ExtendedClass.main()+8
```

Администратор тестов с интервалами: Администратор тестов с интервалами предоставляет тестовые точки, которые активируются с интервалами, определенными пользователем. Тестовые точки не располагаются в коде ядра или приложения, а задаются на основе событий теста, связанных с интервалами реального времени.

Администратор тестов с интервалами может эффективно применяться для обобщения статистики за определенный интервал. Он принимает 4- или 5-кортежную спецификацию тестов в следующем формате:

```
@@interval:*:clock:<# milliseconds>:[*|cpu_ids]
```

Администратор тестов с интервалами фильтрует события теста по ИД процесса, если он задан во втором поле. Если второму полю присвоить значение *, то тест будет активироваться для всех процессов. Кроме того, единственным значением, поддерживаемым администратором тестов с интервалами в третьем поле, является ключевое слово clock, указывающее, что спецификация теста предназначена для теста с реальным временем. В четвертом поле, т. е. в поле <число-миллисекунд>, указывается число миллисекунд между активациями теста. В соответствии с требованиями администратора тестов с интервалами, значение этого поля должно состоять только из цифр от 0 до 9.

В пятом поле, <cpu_ids>, указаны идентификаторы процессоров, для которых запущен тест. Значением этого поля могут быть диапазон идентификаторов процессоров, * или отдельные идентификаторы процессоров. ИД процесса, заданный во втором кортеже, и идентификатор процессора, указанный в пятом кортеже, взаимно исключают друг друга. Поэтому администратор тестов с интервалами фильтрует события теста или по указанному во втором кортеже ИД процесса, работающего на любом процессоре, или по всем процессам, работающим на указанных процессорах. Минимальное поддерживаемое значение времени - 100 миллисекунд. Можно запускать только один тест с интервалами, связанный с процессором, на одном логическом процессоре в системе. Это необязательное поле.

Для тестов с интервалами, в которых не указан идентификатор процесса, значение интервала должно быть кратно 100. Таким образом, для тестов, отличных от интервальных тестов профилирования, допустимы события, отстоящие друг от друга на 100 мс, 200 мс, 300 мс и т. д. В тестах с интервалами, для которых указан идентификатор процесса, интервал должны быть не меньше допустимого интервала для глобального

пользователя root и кратно 10 для других пользователей. Таким образом, для обычных пользователей допустимы события теста, отстоящие друг от друга на 10 мс, 20 мс, 30 мс и т. д. Для одного процесса допустимо только одно значение интервала.

Ниже приведены примеры тестов с интервалами, связанных с процессорами:

1. Для того чтобы получить информацию о контексте, выполните следующий тест ProbeVue на всех процессорах (“*” в поле cpu_ids обозначает все процессоры).

   ```
   @@interval:*:clock:100:*
   ```

2. Для того чтобы получить информацию о контексте для процессоров в диапазоне с 10 по 20, укажите формат x-y, где x=10 и y=20 обозначают диапазон процессоров.

   ```
   @@interval:*:clock:100:10-20
   ```

3. Для того чтобы получить информацию о контексте для процессоров 10 и 12, укажите идентификаторы процессоров, разделенные символом ‘|’ (черта). Этот формат используется для указания больше чем одного процессора.

   ```
   @@interval:*:clock:100:10|12
   ```

4. Для того чтобы получить информацию о контексте для процессоров 10 и с 12 по 20, укажите следующее.

   ```
   @@interval:*:clock:100:10|12-20
   ```

Примечание: Пользователь может указать значение “*” или набор идентификаторов процессоров.

Примечание: Администратор тестов с интервалами не гарантирует, что тест будет активироваться с интервалом, точно соответствующим значению в миллисекундах, указанному в четвертом поле. Прерывания с более высоким приоритетом и код, выполняющийся после выключения всех прерываний, могут привести к более поздней активации теста, чем указано в спецификации.

Администратор тестов с интервалами требует только базовые привилегии динамической трассировки. Администратор тестов с интервалами вводит следующие ограничения на число поддерживаемых тестов для предотвращения злонамеренных попыток пользователей вызвать нехватку памяти для ядра путем создания огромного числа тестов с интервалами.

Таблица 27. Ограничения, установленные администратором тестов с интервалами

Интервал	Число
Максимальное число тестов с интервалами на пользователя	32
Максимальное число тестов с интервалами в системе	1024

Администратор тестов с интервалами не поддерживает следующие функции. Если использовать эти функции в точке теста администратора тестов с интервалами, они будут возвращать пустую строку или ноль.

- **get_function**
- **get_probe**
- **get_location_point**

Если не указан ИД процесса, то тест с интервалами может активироваться в контексте любого процесса, в зависимости от момента активации теста, поскольку событие теста зависит от реального времени. Поэтому среда ProbeVue не разрешает использование любых из следующих функций внутри блока действий администратора тестов с интервалами, чтобы предотвратить несанкционированный доступ к внутренним данным процесса. Это нарушение защиты обрабатывается только в ядре. Сценарий Vue будет успешно скомпилирован, но инициализация сеанса выполнена не будет.

- **stktrace**
- **get_userstring**

Эти функции не возвратят никакого значения, если будут вызваны из этого администратора тестов. Даже пользователь root не может вызвать эти функции из администратора тестов с интервалами.

Когда указан ИД процесса, активируется тест с интервалами для всех нитей в процессе, в указанном интервале времени. Поскольку тест активируется в контексте процесса, то для блока действий администратора тестов с интервалами разрешена функция **stktrace()** и встроенная переменная **__pname**.

Администратор тестов трассировки системы

Администратор тестов трассировки системы предоставляет тестовые точки в любых местах, где могут оказаться перехватчики трассировки системы, предназначенные для трассировки канала системных событий (канала 0), как в ядре, так и в приложениях. Для использования этого администратора тестов необходимы права доступа к ядру; при этом пользователь не должен работать с WPAR.

Администратор тестов трассировки системы принимает 3-кортежную спецификацию тестов в следующем формате:

```
@@systrace:*:<hookid>
```

где параметр *hookid* указывает ИД требуемого перехватчика трассировки системы. Параметр *hookid* состоит из 4 шестнадцатеричных знаков, обычно вида `hhh0`. Например, чтобы задать параметр *hookid* для системного вызова **fork**, укажите 1390. Смотрите примеры в файле **/usr/include/sys/trchkid.h**, такие как HKWD_SYSC_FORK. Записи в этом файле представляют собой слово с данными о событии трассировки, где значение *hookid* содержится в верхнем полуслове. Поскольку слова с данными о событии могут быть случайными, не предусмотрено никакой проверки параметра *hookid*, кроме проверки того, что это допустимая шестнадцатеричная строка, в которой не более 4 шестнадцатеричных знаков. Не будет считаться ошибкой, если указать значение *hookid*, которое никогда не встречается.

Для удобства можно указать параметр *hookid*, содержащий менее 4 шестнадцатеричных знаков. В этом случае прежде всего подразумевается конечный ноль, а затем дополнительные начальные нули в количестве, необходимом для неявного определения требуемых 4 знаков. Например, можно использовать `139` как сокращенный вариант значения `1390`. Аналогично, `0100`, `010`, и `10` равным образом означают одно и то же значение *hookid*, взятое из HKWD_USER1.

Можно указать параметр *hookid*, содержащий символ подстановки *. В этом случае будут тестироваться все системные трассировки, что, скорее всего, приведет к неприемлемому снижению производительности. Поэтому такую спецификацию следует использовать только при безусловной необходимости.

Второй кортеж зарезервирован и может быть задан символом *, в соответствии с описанием.

Тесты активируются только событиями трассировки, которые действительно происходят и записывают данные трассировки системы. В частности, тест трассировки системы может произойти только при активной трассировке системы. Администратор тестов systrace является администратор тестов на основе событий. Поэтому имя теста, имя функции, и точка расположения недоступны. Поскольку слово с данными о событии трассировки передается в сценарий, это ограничение не имеет существенного значения.

Для пользователя, отличного от root, установлено ограничение - он может одновременно включить не более 64 тестов systrace. Во всей системе можно включить не более 128 явных тестов systrace.

Встроенные регистровые переменные ProbeVue позволяют обращаться к данным трассировки. Для этой цели нельзя использовать переменные __arg*. Существует два общих стиля трассировки системы.

Следующий стиль предназначен для перехватчиков trchook(64)/utrchook(64) (или эквивалентные макросы TRCHKLx в C):

- __r3 содержит 16-разрядный hookid.
- __r4 содержит subhookid.
- __r5 содержит слово D1 с данными трассировки.
- __r6 содержит слово D2 с данными трассировки.
- __r7 содержит слово D3 с данными трассировки.

- __r8 содержит слово D4 с данными трассировки.
- __r9 содержит слово D5 с данными трассировки.

Не все перехватчики трассировки содержат все 5 слов с данными. Слова с неопределенными данными из заданного перехватчика трассировки отображаются нулевым значением. Блоку Vue для заданного ИД перехватчика должна быть доступна информация о том, какие данные и в каком количестве трассируются перехватчиком с этим ИД.

Если запись трассировки была сделана одной из функций семейства trcgen или trcgent, используйте следующий стиль:

- __r3 содержит 16-разрядный hookid.
- __r4 содержит subhookid.
- __r5 содержит слово D1 с данными трассировки.
- __r6 содержит длину данных трассировки.
- __r7 содержит адрес данных трассировки.

В следующем сценарии показан простой пример администратора тестов systrace:

```
@@systrace:*:1390
{
 if (__r4 == 0) { /* normal fork is traced with subhookid zero */
  printf("HKWD_SYSC_FORK: %d forks child %d\n", __pid, __r5);
  exit();
 }
}
```

Администратор тестов Systrace работает независимо от функции трассировки системы и поддерживает трассировку расположений перехватчиков даже в том случае, если функция трассировки системы неактивна. Функцию трассировки системы можно включить в сеансе ProbeVue.

Администратор тестов Systrace может использовать ядро операционных систем AIX. Отслеживание перехватчиков, применяемых ProbeVue, может привести к снижению надежности AIX. Таким образом, в процессе трассировки таких ИД перехватчиков нельзя использовать отдельные конструкции ProbeVue. В следующей таблице перечислены исключения:

Примечание: Следующие конструкции игнорируются в случае трассировки всех перехватчиков с помощью спецификации `@@systrace:*:*`. Трассировка стека можно быть недоступна для просмотра, если ядро AIX запрещает создание исключительных ситуаций в среде, в которой расположен перехватчик. Возможность отображения данных трассировки стека с помощью ProbeVue определяется во время выполнения.

Таблица 28. Конструкции точек трассировки

Номер	Точка трассировки	Конструкция
1	HKWD_KERN_HCALL	ALL
2	HKWD_KERN_SLIH	Именованный массив, диапазон, переменная stktrace, __stat
3	HKWD_KERN_LOCK	Именованный массив, диапазон, переменная stktrace, __stat
4	HKWD_KERN_UNLOCK	Именованный массив, диапазон, переменная stktrace, __stat
5	HKWD_KERN_DISABLEMENT	ALL
6	HKWD_KERN_DISPATCH	__ublock, stktrace, get_stktrace, __pname, __execname, __errno
7	HKWD_KERN_DISPATCH_SRAD	__ublock, stktrace, get_stktrace, __pname, __execname, __errno
8	HKWD_KERN_DISP_AFFIN	__ublock, stktrace, get_stktrace, __pname, __execname, __errno
9	HKWD_KERN_UNDISP	__ublock, stktrace, get_stktrace, __pname, __execname, __errno
10	HKWD_KERN_IDLE	__ublock, stktrace, get_stktrace, __pname, __execname, __errno
11	HKWD_KERN_FLIH	Именованный массив, диапазон, переменная stktrace, __stat
12	HKWD_KERN_RESUME	Именованный массив, диапазон, переменная stktrace, __stat

Таблица 28. Конструкции точек трассировки (продолжение)

Номер	Точка трассировки	Конструкция
13	HKWD_KERN_VPM	Именованный массив, диапазон, переменная stktrace, __stat
14	HKWD_PM_NOTIFY	Именованный массив, диапазон, переменная stktrace, __stat

При наличии соответствующих прав доступа сценарий Vue может сам создать записи трассировки системы с помощью функций событий RAS. Однако администратор тестов Systrace не распознает записи трассировки, созданные с помощью сценария Vue.

Администратор тестов расширенных системных вызовов (syscallx): Администратор тестов syscallx позволяет отследить все системные вызовы. Базовые системные вызовы - это набор системных вызовов, экспортируемый ядром и его базовыми расширениями, и доступный сразу после загрузки. Не поддерживаются системные вызовы, загружаемые уже после загрузки системы. В кортеже тестовой точки может быть указан отдельный системный вызов или же все вызовы. В отличие от администратора тестов syscall, третье поле кортежа тестовой точки для syscallx должно указывать фактическую функцию точки входа ядра. Если ИД процесса указан во втором поле кортежа тестовой точки, то администратор тестов syscallx ограничивает число активируемых тестов для данного процесса.

Примеры:

```
/* Кортеж тестовой точки для тестирования системного вызова read для всех процессов */
@@syscallx:*:kread:entry

/* Кортеж тестовой точки для тестирования системного вызова fork для процесса 434 */
@@syscallx:434:kfork:exit

/* Кортеж тестовой точки для тестирования точки входа всех системных вызовов */
@@syscallx:*:*:entry

/* Кортеж тестовой точки для тестирования точки входа всех системных вызовов для процесса 744 */
@@syscallx:744:*:exit
```

Системные вызовы, поддерживаемые администратором тестов syscall

В следующей таблице перечислены системные вызовы, поддерживаемые администратором тестов syscall, вместе с фактическими именами точек входа ядра.

Примечание: Имя точки входа ядра приводится здесь только в целях документации. Имена точек входа ядра могут отличаться в разных выпусках или даже изменяться после служебного обновления.

Таблица 29. Системные вызовы, поддерживаемые администратором тестов syscall

Имя системного вызова	Имя точки входа ядра
absinterval	absinterval
accept	accept1
bind	bind
close	close
creat	creat
execve	execve
exit	_exit
fork	kfork
getgidx	getgidx
getgroups	getgroups
getinterval	getinterval
getpeername	getpeername
getpid	_getpid
getppid	_getppid
getpri	_getpri

Таблица 29. Системные вызовы, поддерживаемые администратором тестов syscall (продолжение)

Имя системного вызова	Имя точки входа ядра
getpriority	_getpriority
getsockname	getsockname
getsockopt	getsockopt
getuidx	getuidx
incinterval	incinterval
kill	kill
listen	listen
lseek	klseek
mknod	mknod
mmap	mmap
mq_close	mq_close
mq_getattr	mq_getattr
mq_notify	mq_notify
mq_open	mq_open
mq_receive	mq_receive
mq_send	mq_send
mq_setattr	mq_setattr
mq_unlink	mq_unlink
msgctl	msgctl
msgget	msgget
msgrcv	__msgrcv
msgsnd	__msgsnd
nsleep	_nsleep
open	kopen
pause	_pause
pipe	pipe
plock	plock
poll	_poll
read	kread
reboot	reboot
recv	_erecv
recvfrom	_enrecvfrom
recvmsg	_erecvmsg
select	_select
sem_close	_sem_close
sem_destroy	sem_destroy
sem_getvalue	sem_getvalue
sem_init	sem_init
sem_open	_sem_open
sem_post	sem_post
sem_unlink	sem_unlink
sem_wait	_sem_wait
semctl	semctl
semget	semget
semop	__semop

Таблица 29. Системные вызовы, поддерживаемые администратором тестов syscall (продолжение)

Имя системного вызова	Имя точки входа ядра
semtimedop	__semtimedop
send	_esend
sendmsg	_esendmsg
sendto	_esendto
setpri	_setpri
setpriority	_setpriority
setsockopt	setsockopt
setuidx	setuidx
shmat	shmat
shmctl	shmctl
shmdt	shmdt
shmget	shmget
shutdown	shutdown
sigaction	_sigaction
sigpending	_sigpending
sigprocmask	sigprocmask
sigsuspend	_sigsuspend
socket	socket
socketpair	socketpair
stat	statx
waitpid	kwaitpid
write	kwrite

Запуск в WPAR

Разделы рабочей схемы (WPAR) - это виртуальные среды операционной системы, работающие под управлением одного экземпляра реальной операционной системы AIX. Среда WPAR некоторым образом отличается от стандартной среды операционной системы AIX.

В среде WPAR поддерживается динамическая трассировка. По умолчанию при создании WPAR этому разделу WPAR будут присвоены только привилегии **PV_PROBEVUE_TRC_USER_SELF** и **PV_PROBEVUE_TRC_USER**, причем эти права доступа будут предоставлены администратору (root) системы WPAR. Пользователь admin из глобального раздела может изменить значение набора привилегий WPAR, задаваемое по умолчанию, или может явным образом присвоить дополнительные права доступа при создании WPAR.

Привилегии в разделе WPAR обычно имеют такое же смысловое значение, как и в глобальном разделе. Следует проявлять осторожность при присвоении привилегий **PV_PROBEVUE_TRC_KERNEL** или **PV_PROBEVUE_TRC_MANAGE** разделу WPAR. Любой пользователь с правами доступа **PV_PROBEVUE_TRC_KERNEL** может обратиться к глобальным переменным ядра, а пользователь с правами доступа **PV_PROBEVUE_TRC_MANAGE** может изменить значения параметров ProbeVue или завершить работу ProbeVue. Эти изменения отразятся на всех пользователях, даже из других разделов.

При вводе команды **probevue** в разделе WPAR в нем будут невидимы процессы, запущенные в других WPAR или в глобальном разделе. Поэтому можно тестировать процессы только в собственном WPAR. Команда **probevue** не будет выполнена, если в спецификации теста содержится ИД процесса, выполняемого за пределами данного раздела. Права доступа **PV_PROBEVUE_TRC_USER** и **PV_PROBEVUE_TRC_SYSCALL** в разделе WPAR позволяют тестировать только функции пользовательского пространства или системные вызовы процессов, выполняемых в собственном WPAR. При

тестировании системных вызовов необходимо во втором поле спецификации теста syscall указать допустимый ИД процесса, видимого в разделе WPAR. Использование значения * во втором поле не поддерживается.

Если сеанс ProbeVue инициирован в мобильном WPAR, то выполняется временное переключение WPAR в состояние, не допускающее использование контрольных точек. После завершения сеанса ProbeVue в разделе WPAR опять можно использовать контрольные точки.

Администратор тестов ввода-вывода:
Администратор тестов ввода-вывода позволяет трассировать события операций ввода-вывода на различных уровнях стека ввода-вывода AIX. Администратор тестов syscall используется для трассировки запроса ввода-вывода приложения, который активируется системным вызовом read/write. Администратор тестов ввода-вывода используется для углубленного тестирования на уровне syscall.

Администратор тестов ввода-вывода используется для анализа времени ответа операций ввода-вывода блочного устройства, которое разделяет время обработки и задержку очереди.

Поддерживаются следующие уровни:
- Логическая файловая система (LFS)
- Виртуальная файловая система (VFS)
- Расширенная журнализированная файловая система (JFS2)
- Logical Volume Manager (LVM)
- Драйвер диска SCSI
- Общие блочные устройства

Основные сферы применения администратора тестов ввода-вывода:
- Обнаружение следующих закономерностей ввода-вывода устройства. Допустимыми устройствами могут быть диск, логический том, группа томов или файловая система (тип или путь к точке монтирования) за указанный период времени:
  - Число операций ввода-вывода
  - Размер операций ввода-вывода
  - Тип операции ввода-вывода (чтение/запись)
  - Последовательный или случайный характер ввода-вывода
- Получение информации об использовании файловой системы (тип или путь к точке монтирования), логического тома, группы томов или диска на уровне процесса или нити.
- Получение сквозного соответствия потока ввода-вывода на различных уровнях (где это возможно).
- Мониторинг использования определенного ресурса ввода-вывода. Пример:
  - Трассировка всех операций записи файла `/etc/password`.
  - Трассировка операции чтения блока 0 устройства `hdisk0`.
  - Трассировка при открытии нового логического тома в корневой группе томов (`rootvg`).
- Для дисков MPIO (разветвленный ввод-вывод) получение информации о путях посредством следующих действий:
  - Получение информации об использовании и времени ответа уровня пути.
  - Обнаружение смены пути или сбоя пути.
- Для ошибок ввода-вывода получение дополнительной информации об ошибке на уровне драйвера диска.

Спецификация теста

Тесты ввода-вывода должны указываться в сценарии Vue в следующем формате:
`@@io:`*`подтип`*`:`*`событие-ввода-вывода`*`:`*`тип-операции`*`:`*`фильтр`*`[|`*`фильтр ...`*`]`

Эта спецификация состоит из 5 кортежей, разделенных двоеточием (`:`). Первый кортеж всегда `@@io`.

Подтип теста

Второй кортеж обозначает подтип теста, указывающий уровень стека ввода-вывода AIX, содержащий тест. Этот кортеж может иметь одно из следующих значений:

Таблица 30. Второй кортеж теста

Второй кортеж (подтип)	Описание
disk	Этот тест запускается для событий драйвера диска. В настоящее время администратор тестов ввода-вывода поддерживает только драйвер `scsidisk`.
lvm	Этот тест запускается для событий администратора логических томов (LVM).
bdev	Этот тест запускается для любого устройства блочного ввода-вывода. Примеры блочных устройств: жесткий диск, привод компакт-дисков или дискет. Этот подтип используется, только когда ни один из других подтипов не применим. Например, если блочное устройство не диск, группа томов или логическим том, то этот подтип применим.
jfs2	Этот тест запускается для событий файловой системы JFS2.
vfs	Этот тест запускается для любых операций файлового ввода-вывода.

Примечание: Второй кортеж не может быть звездочкой (*).

Для типа disk во втором кортеже третий кортеж может иметь следующие значения:

Таблица 31. Значения третьего кортежа, когда второй кортеж - disk

Подтип (второй кортеж)	Событие ввода-вывода (третий кортеж)	Описание
disk	entry	Этот тест запускается, когда драйвер диска получает запрос ввода-вывода для обработки.
	iostart	Этот тест запускается, когда драйвер диска выбирает запрос ввода-вывода из своей очереди готовых запросов и передает его на нижний уровень (например, драйверу адаптера). Одиночный исходный запрос ввода-вывода к драйверу диска может передавать множественные командные запросы (некоторые могут быть командными запросами управления задачами, связанными с драйвером) на нижний уровень. Однако иногда драйвер может объединять несколько исходных запросов в один запрос и передавать его на нижний уровень.
	iodone	Этот тест запускается, когда нижний уровень, например драйвер адаптера, возвращает запрос ввода-вывода (успешный или нет) драйверу диска.
	exit	Этот тест запускается, когда драйвер диска возвращает запрос ввода-вывода (успешный или нет) на верхний уровень.

Примечание: Элементы значений следующих встроенных переменных доступны в тестах, указанных для подтипа теста: `__iobuf`, `__diskinfo`, `__diskcmd` (только в `disk:iostart` и `disk:iodone`) и `__iopath` (только в `disk:iostart` и `disk:iodone`).

Для каждого теста entry определен соответствующий тест exit, имеющий одинаковое значение `__iobuf->bufid` в обеих точках тестирования. За событием entry может следовать несколько событий `iostart`, но хотя бы одно событие должно иметь такое же значение `__iobuf->bufid`. Каждое событие `iostart` имеет соответствующее событие `iodone` с таким же значением `__iobuf->child_bufid`.

Для типа LVM во втором кортеже третий кортеж может иметь следующие значения:

Таблица 32. Значения третьего кортежа, когда второй кортеж - LVM

Подтип (второй кортеж)	Событие ввода-вывода (третий кортеж)	Описание
lvm	entry	Этот тест запускается, когда уровень LVM получает запрос ввода-вывода для обработки.
	iostart	Этот тест запускается, когда LVM выбирает запрос ввода-вывода из своей очереди готовых запросов и передает его на нижний уровень (обычно это драйвер диска).
	iodone	Этот тест запускается, когда нижний уровень, например драйвер диска, возвращает запрос ввода-вывода (успешный или нет) LVM.
	exit	Этот тест запускается, когда LVM возвращает запрос ввода-вывода (успешный или нет) на верхний уровень.

Примечание: Элементы значений следующих встроенных переменных доступны в тестах, указанных в LVM: `__iobuf`, `__lvol` и `__volgrp`. Каждый тест entry имеет соответствующий тест exit, имеющий одинаковое значение `__iobuf->bufid` в обеих точках тестирования.

За событием entry может следовать несколько событий `iostart`, но хотя бы одно событие должно иметь такое же значение `__iobuf->bufid`. Каждое событие `iostart` имеет соответствующее событие `iodone` с таким же значением `__iobuf->child_bufid`.

Для тестов общих блочных устройств третий кортеж может иметь следующие значения:

Таблица 33. Значения третьего кортежа, когда второй кортеж - общее блочное устройство

Подтип (второй кортеж)	Событие ввода-вывода (третий кортеж)	Описание
bdev	iostart	Этот тест активируется, когда начинает работу одно из устройств блочного ввода-вывода, например жесткий диск, логический том или привод компакт-дисков. Это происходит при вызове службы ядра AIX `devstrat` любым кодом.
	iodone	Этот тест активируется, когда завершается выполнение запроса блочного ввода-вывода вызовом службы ядра AIX `iodone` любым кодом.

Примечание: Элементы значений следующей встроенной переменной доступны в тестах, указанных в bdev: `__iobuf`. Каждое событие `iostart` имеет соответствующее событие `iodone` с таким же значением `__iobuf->bufid`.

Для тестов файловой системы JFS2 третий кортеж может иметь следующие значения:

Таблица 34. Значения третьего кортежа, когда второй кортеж - JFS2

Подтип (второй кортеж)	Событие ввода-вывода (третий кортеж)	Описание
jfs2	buf_map	Этот тест запускается, когда область логического файла отображается в буфер ввода-вывода и отправляется в соответствующий логический том.

Примечание: Элементы значений следующей встроенной переменной доступны в тесте файловой системы JFS2: `__j2info`.

Для тестов виртуальной файловой системы (VFS) третий кортеж может иметь следующие значения:

Таблица 35. Значения третьего кортежа, когда второй кортеж - VFS

Подтип (второй кортеж)	Событие ввода-вывода (третий кортеж)	Описание
vfs	entry	Этот тест запускается в начале выполнения любой операции файлового ввода-вывода.
	exit	Этот тест запускается при завершении любой операции файлового ввода-вывода (вне зависимости от ее успешности).

Примечание: Элементы следующей встроенной переменной доступны в тесте VFS: `__file`.

В рамках одной нити за каждым событием entry следует событие exit с таким же значением `__file->inode_id`.

Тип операции теста

Четвертый кортеж содержит тип операции ввода-вывода, указанной в тесте. Четвертый кортеж может иметь одно из следующих значений:

Таблица 36. Четвертый кортеж для операции ввода-вывода

Четвертый кортеж	Описание
read	Тест запускается только для операции чтения.
write	Тест запускается только для операции записи.
*	Тест запускается и для операций чтения, и для операций записи.

Фильтр тестов

Пятый кортеж - это кортеж фильтра, позволяющий фильтровать тесты в соответствии с требованием. Допустимые значения зависят от подтипа. Можно указывать несколько значений через символ |. В этом случае тест запускается, если соответствует любому из указанных фильтров. Если значение пятого кортежа - *, фильтрация не выполняется, и тест запускается, если совпадают остальные кортежи. Если указано несколько селекторов и один из них - *, это равносильно указанию значения * для всего кортежа.

Для тестов диска пятый кортеж может иметь следующие значения:

Таблица 37. Кортеж фильтра для дисков

Фильтр (пятый кортеж)	Описание
Имя диска. Например, `hdisk0`	Действие теста выполняется только для определенного диска.
Тип диска. Допустимые символы: FC, ISCSI, VSCSI, SAS	Действие теста выполняется только для дисков определенного типа. Расшифровка символов: • FC - диск Fibre Channel • ISCSI - диск iSCSI • VSCSI - виртуальный диск SCSI (на клиенте VIOS) • SAS - последовательно подключаемые диски SCSI

Примечание: Фильтры по имени и типу диска можно сочетать. Например, следующий тест запускается для любого диска `hdisk0` или любого другого диска Fibre Channel (по событию entry диска для операций чтения и записи)

```
@@io:disk:entry:*:hdisk0|FC
```

Для тестов администратора логических томов (LVM) пятый кортеж может иметь следующие значения:

Таблица 38. Кортеж фильтра LVM

Фильтр (пятый кортеж)	Описание
Имя логического тома, например `hd5, lg_dumplv`	Действие теста выполняется только для определенного логического тома.
Имя группы томов, например `rootvg`	Действие теста выполняется только для логических томов, принадлежащих определенной группе томов.

Следующий тест запускается для любого логического тома, принадлежащего корневой группе томов (rootvg) или тестовой группы томов (testvg) (по событию iostart только операции записи):

```
@@io:lvm:iostart:write:rootvg|testvg
```

Для тестов общих блочных устройств пятый кортеж может иметь следующие значения:

Таблица 39. Кортеж фильтра общего блочного устройства

Фильтр (пятый кортеж)	Описание
Имя блочного устройства, например: `hdisk0, hd5, cd0`	Действие теста выполняется только для определенного блочного устройства.

Примеры для тестов общих блочных устройств:

```
@@io:bdev:iostart:*:cd0
```

```
@@io:bdev:iodone:read:hdisk3|hdisk5
```

Для тестов файловой системы JFS2 пятый кортеж может иметь следующие значения:

Таблица 40. Кортеж фильтра JFS2

Фильтр (пятый кортеж)	Описание
Путь к точке монтирования файловой системы, например `/usr`	Действие теста выполняется только для файловой системы с определенным путем к точке монтирования. Это должна быть файловая система JFS2. В противном случае ProbeVue отклонит спецификацию теста.

Примеры для тестов файловой системы JFS2:

```
@@io:jfs2:buf_map:*:/usr|/tmp
```

Для тестов виртуальной файловой системы (VFS) пятый кортеж может иметь следующие значения:

Таблица 41. Кортеж фильтра VFS

Фильтр (пятый кортеж)	Описание
Путь к точке монтирования файловой системы. Например, `/tmp`	Действие теста выполняется для файлов, принадлежащих данной файловой системе.
Тип файловой системы. Допустимые символы: `JFS2, NAMEFS, NFS, JFS, CDROM, PROCFS, SFS, CACHEFS, NFS3, AUTOFS, POOLFS, VXFS, VXODM, UDF, NFS4, RFS4, CIFS, PMEMFS, AHAFS, STNFS, ASMFS`	Действие теста выполняется для файлов определенной файловой системы. Символы соответствуют файловым системам AIX, определенным в экспортированном заголовочном файле `sys/vmount.h`.

Примеры для тестов виртуальной файловой системы (VFS):

```
@@io:vfs:entry:read:JFS2
```

```
@@io:vfs:exit:*:/usr|JFS
```

Встроенные переменные для сценариев Vue, связанные с тестами ввода-вывода

Встроенная переменная `__iobuf`

Можно использовать встроенную переменную `__iobuf` для доступа к различной информации о буфере ввода-вывода, который используется в текущей операции ввода-вывода. Она доступна в тестах следующих подтипов: `disk`, `lvm` и `bdev`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__iobuf->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как `Invalid Value` (недопустимое значение). Это значение возвращается по одной из следующих причин:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная `__iobuf` содержит следующие элементы:

Таблица 42. Элементы встроенной переменной __iobuf

Имя элемента	Тип	Описание	Недопустимое значение
blknum	unsigned long long	Номер начального блока запроса ввода-вывода.	0xFFFFFFFFFFFFFFFF
bcount	unsigned long long	Запрашиваемое количество байтов в операции ввода-вывода.	0xFFFFFFFFFFFFFFFF
bflags	unsigned long long	Флаги, связанные с операцией ввода-вывода. Доступные символы: `B_READ`, `B_ASYNC`, `B_ERROR`. Эти символы могут использоваться для проверки состояния соответствующего флага в значении bflags. Например, если выражение (`__iobuf->bflags & B_READ`) истинное, то это операция чтения. **Примечание:** Флага `B_WRITE` не существует. Если флаг `B_READ` не установлен, значит операция является операцией записи.	0
devnum	unsigned long long	Номер целевого устройства, связанного с операцией ввода-вывода. Он состоит из основного номера и дополнительного номера.	0
major_num	int	Основной номер целевого устройства операции ввода-вывода.	-1
minor_num	int	Дополнительный номер целевого устройства операции ввода-вывода.	-1
error	int	Номер ошибки в случае ошибки операции ввода-вывода. Это значение определено в экспортированном заголовочном файле `errno.h`.	-1

Таблица 42. Элементы встроенной переменной __iobuf (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
residue	unsigned long long	Оставшееся количество байтов из исходного запроса, которые не удалось прочитать или записать. В событиях завершения ввода-вывода это значение всегда нулевое. Однако для операции чтения ненулевое значение может говорить о попытке прочитать больше, чем есть; что допустимо. Это значение учитывается, только когда значение ошибки ненулевое.	0xFFFFFFFFFFFFFFFF
bufid	unsigned long long	Уникальный номер, связанный с запросом ввода-вывода. В процессе выполнения ввод-вывода значение bufid уникально идентифицирует запрос ввода-вывода во всех событиях определенного подтипа. Например, в событиях `disk: entry`, `disk: iostart`, `disk: iodone` и `disk:exit`. Если `__iobuf->bufid` совпадает, значит это тот же запрос ввода-вывода на разных этапах.	0
parent_bufid	unsigned long long	Если значение ненулевое, то это `bufid` буфера верхнего уровня, связанного с данным запросом ввода-вывода. Теперь можно связать текущую операцию ввода-вывода с запросом ввода-вывода верхнего уровня. Например, в запросе дискового ввода-вывода можно определить соответствующий ввод-вывод LVM. **Примечание:** Значение поля parent_bufid задается не всегда, поэтому оно не всегда полезно. Рекомендуется использовать поле child_bufid для установки связи запросов ввода-вывода между двумя соседними уровнями.	0
child_bufid	unsigned long long	Если значение ненулевое, то это `bufid` нового запроса ввода-вывода, переданного на нижний уровень. Наиболее подходящие события для записи: `disk:iostart`, `lvm:iostart` `bdev:iostart`. Можно идентифицировать ввод-вывод на соседнем нижнем уровне путем сравнения значения `__iobuf->bufid` с данным значением child_bufid. Например, в `lvm:iostart` можно записать значение `__iobuf->child_buf`. Затем в `disk:entry` можно сравнить его с `__iobuf->bufid` и определить соответствующий запрос ввода-вывода.	0

Встроенная переменная **__file**

Можно использовать специальную встроенную переменную `__file` для получения различной информации о файловой операции. Она доступна в тестах подтипа VFS (виртуальная файловая система). К ее элементам можно обращаться с помощью следующего синтаксиса: `__file->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как недопустимое. Недопустимое значение возвращается по одной из следующих причин:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная `__file` содержит следующие элементы:

Таблица 43. Элементы встроенной переменной __file

Имя элемента	Тип	Описание	Недопустимое значение
f_type	int	Тип файла. Может совпадать с одним из следующих значений встроенной константы: • F_REG (обычный файл) • F_DIR (каталог) • F_BLK (файл блочного устройства) • F_CHR (файл символьного устройства) • F_LNK (ссылка на файл) • F_SOCK (сокет) **Примечание:** Значение может не совпадать со встроенными константами, поскольку этот список не включает все возможные типы файлов, но только самые полезные.	-1

Таблица 43. Элементы встроенной переменной __file (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
fs_type	int	Тип файловой системы, которой принадлежит данный файл. Может совпадать с одним из следующих значений встроенной константы: • FS_JFS2 • FS_NAMEFS • FS_NFS • FS_JFS • FS_CDROM • FS_PROCFS • FS_SFS • FS_CACHEFS • FS_NFS3 • FS_AUTOFS • FS_POOLFS • FS_VXFS • FS_VXODM • FS_UDF • FS_NFS4 • FS_RFS4 • FS_CIFS • FS_PMEMFS • FS_AHAFS • FS_STNFS • FS_ASMFS Встроенные константы соответствуют типам файловой системы AIX из экспортированного файла заголовка `sys/vmount.h`.	-1
mount_path	char *	Путь к точке монтирования связанной файловой системы.	пустая строка
devnum	unsigned long long	Номер связанного блочного устройства файла. Состоит из основного номера и дополнительного номера. Если связанного блочного устройства нет, то это значение равно нулю.	0
major_num	int	Основной номер связанного блочного устройства файла.	-1
minor_num	int	Дополнительный номер связанного блочного устройства файла.	-1
offset	unsigned long long	Текущее байтовое смещение ввода-вывода в файле.	0xFFFFFFFFFFFFFFFF
rw_mode	int	Режим ввода-вывода файла. Совпадает с одним из значений встроенной константы: F_READ или F_WRITE.	-1

Таблица 43. Элементы встроенной переменной __file (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
byte_count	unsigned long long	В событии entry для `vfs:` `byte_count` содержит количество байтов запроса чтения или записи. В событии exit для `vfs:` он содержит количество байтов, оставшихся необработанными. Например, разность этих значений у обоих событий показывает, сколько байтов обработано операцией.	0xFFFFFFFFFFFFFFFF
fname	char *	Имя файла (только имя, не путь).	пустая строка
inode_id	unsigned long long	Общесистемный уникальный номер, связанный с файлом. **Примечание:** Это не номер I-узла файла.	0
path	path_t (новый тип данных в языке VUE)	Полный путь к файлу. Его можно выводить функцией `printf()` с помощью спецификатора формата `%p`.	пустая строка как путь к файлу
error	int	В случае ошибки операции чтения/записи содержит номер ошибки, определенный в экспортированном заголовочном файле `errno.h`. Если ошибок нет, имеет нулевое значение.	-1

Встроенная переменная **__lvol**

Можно использовать специальную встроенную переменную `__lvol` для получения различной информации о логическом томе в операции LVM. Она доступна в тестах подтипа `lvm`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__lvol->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как `Invalid Value` (недопустимое значение). Это недопустимое значение может возвращаться по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная `__lvol` состоит из следующих элементов:

Таблица 44. Элементы встроенной переменной __lvol

Имя элемента	Тип	Описание	Недопустимое значение
name	char *	Имя логического тома.	пустая строка
devnum	unsigned long long	Номер устройства логического тома. Он состоит из основного номера и дополнительного номера.	0
major_num	int	Основной номер логического тома.	-1
minor_num	int	Дополнительный номер логического тома.	-1

Таблица 44. Элементы встроенной переменной __lvol (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
lv_options	unsigned int	Опции, связанные с логическим томом. Следующие значения заданы в качестве встроенных констант: • LV_RDONLY (логическим том, доступный только для чтения) • LV_NOMWC (без проверки согласования зеркальных копий при записи) • LV_ACTIVE_MWC (активное согласование зеркальных копий при записи) • LV_PASSIVE_MWC (пассивное согласование зеркальных копий при записи) • LV_SERIALIZE_IO (сериализованный ввод-вывод) • LV_DMPDEV (этот логический том является устройством дампа) Узнать состояние этих опций можно с помощью следующего выражения: `__lvol->lv_options & LV_RDONLY`. **Примечание:** Определены не все возможные значения, поэтому могут быть и другие опции.	0xFFFFFFFF

Встроенная переменная **__volgrp**

Можно использовать специальную встроенную переменную `__volgrp` для получения различной информации о группе томов в операции LVM. Она доступна в тестах подтипа `lvm`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__volgrp->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как `Invalid Value` (недопустимое значение). Значение может быть недопустимым по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная `__volgrp` состоит из следующих элементов:

Таблица 45. Элементы встроенной переменной __volgrp

Имя элемента	Тип	Описание	Недопустимое значение
name	char *	Имя группы томов.	пустая строка
devnum	unsigned long long	Номер устройства группы томов. Он состоит из основного номера и дополнительного номера.	0
major_num	int	Основной номер группы томов.	-1
minor_num	int	Дополнительный номер группы томов. **Примечание:** Для групп томов AIX всегда присваивает нулевой дополнительный номер.	-1
num_open_lvs	int	Число открытых логических томов, принадлежащих данной группе томов.	-1

Встроенная переменная **__diskinfo**

Можно использовать специальную встроенную переменную `__diskinfo` для получения различной информации о диске в операции дискового ввода-вывода. Она доступна в тестах подтипа disk. К ее элементам можно обращаться с помощью следующего синтаксиса: `__diskinfo->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как "недопустимое значение". Это значение может возвращаться по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная `__diskinfo` состоит из следующих элементов:

Таблица 46. Элементы встроенной переменной __diskinfo

Имя элемента	Тип	Описание	Недопустимое значение
name	char *	Имя диска.	пустая строка.
devnum	unsigned long long	Номер устройства диска. Он состоит из основного номера и дополнительного номера.	0
major_num	int	Основной номер диска.	-1
minor_num	int	Дополнительный номер диска.	-1
lun_id	unsigned long long	Номер логического накопителя (LUN) для диска.	0xFFFFFFFFFFFFFFFF
transport_type	int	Тип транспорта диска. Может совпадать с одним из следующих значений встроенной константы: • T_FC (Fibre Channel) • T_ISCSI (iSCSI) • T_VSCSI (виртуальный SCSI) • T_SAS (SCSI с последовательным подключением)	-1

Таблица 46. Элементы встроенной переменной __diskinfo (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
queue_depth	int	Длина очереди диска. Показывает, сколько одновременных запросов ввода-вывода драйвер диска может передавать на нижний уровень (например, в адаптер). Если число входящих запросов ввода-вывода больше значения queue_depth, запрос обрабатывается по-другому. Лишний запрос помещается драйвером диска в очередь ожидания, пока нижний уровень не ответит на хотя бы один запрос ввода-вывода, ожидающий обработки.	-1
cmds_out	int	Число ожидающих обработки запросов ввода-вывода к нижнему уровню, например к адаптеру.	-1
path_count	int	Число путей MPIO (разветвленный ввод-вывод) диска (только если диск поддерживает MPIO, в противном случае 0).	-1
reserve_policy	int	Стратегия резервирования SCSI для диска. Совпадает с одним из следующих значений встроенной константы: • DK_NO_RESERVE (no_reserve) • DK_SINGLE_PATH (single_path) • DK_PR_EXCLUSIVE (PR_exclusive) • DK_PR_SHARED (PR_shared) Дополнительные стратегии резервирования приведены в документации AIX по разветвленному вводу-выводу.	-1

Таблица 46. Элементы встроенной переменной __diskinfo (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
scsi_flags	int	Флаги SCSI диска. Заданы следующие значения флагов встроенных констант: • SC_AUTOSENSE_ENABLED (В случае ошибки целевое устройство возвращает информацию об ошибке в ответе. Инициатору не нужно отправлять команду запроса информации об ошибке.) • SC_NACA_1_ENABLED (Включен обычный ACA и целевое устройство переходит в состояние ACA, если оно возвращает условие проверки.) • SC_64BIT_IDS (64-разрядные ИД SCSI и номер логического накопителя (LUN) • SC_LUN_RESET_ENABLED (Разрешена команда сброса LUN.) • SC_PRIORITY_SUP (Устройство поддерживает приоритет ввода-вывода.) • SC_CACHE_HINT_SUP (Устройство поддерживает подсказки кэша.) • SC_QUEUE_UNTAGGED (Устройство поддерживает постановку бестеговых команд в очередь.) **Примечание:** Не все значения флагов определены, поэтому могут быть и другие флаги.	0

Встроенная переменная **__diskcmd**

Можно использовать специальную встроенную переменную `__diskcmd` для получения различной информации о команде ввода-вывода SCSI для текущей операции. Она доступна в тестах подтипа disk (но только для событий `iostart` и `iodone`). К ее элементам можно обращаться с помощью следующего синтаксиса: `__diskcmd->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как "недопустимое значение". Это значение может возвращаться по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра **num_pagefaults** команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

Встроенная переменная **__diskcmd** содержит следующие элементы:

Таблица 47. Элементы встроенной переменной __diskcmd

Имя элемента	Тип	Описание
cmd_type	int	Тип команды SCSI (тип и подтип объединены). Следующие значения встроенных констант доступны в качестве типа команды: • DK_BUF (обычная операция чтения/записи) • DK_IOCTL (ioctl) • DK_REQSNS (запросить информацию об ошибке) • DK_TGT_LUN_RST (сброс целевого устройства или LUN) • DK_TUR (проверить готовность устройства) • DK_INQUIRY (запрос) • DK_RESERVE (зарезервировать SCSI-2, 6-байтовая версия) • DK_RELEASE (освободить SCSI-2, 6-байтовая версия) • DK_RESERVE_10 (зарезервировать SCSI-2, 10-байтовая версия) • DK_RELEASE_10 (освободить SCSI-2, 10-байтовая версия) • DK_PR_RESERVE (постоянный резерв SCSI-3, зарезервировать) • DK_PR_RELEASE (постоянный резерв SCSI-3, освободить) • DK_PR_CLEAR (постоянный резерв SCSI-3, очистить) • DK_PR_PREEMPT (постоянный резерв SCSI-3, заместить) • DK_PR_PREEMPT_ABORT (постоянный резерв SCSI-3, заместить и прервать) • DK_READCAP (емкость чтения, 10-байтовая версия) • DK_READCAP16 (емкость чтения, 16-байтовая версия) **Примечание:** Встроенные константы представляют собой битовые флаги, поэтому для проверки следует использовать оператор &, а не оператор ==. Пример: `__diskcmd->cmd_type & DK_IOCTL`.
retry_count	int	Показывает, повторяется ли операция ввода-вывода после сбоя. **Примечание:** Значение 1 говорит о том, что это первая попытка. Большее значение указывает на фактические повторные попытки.
path_switch_count	int	Показывает, сколько раз менялся путь для данной операции ввода-вывода (обычно это является признаком временного или постоянного сбоя одного из путей ввода-вывода).

Таблица 47. Элементы встроенной переменной __diskcmd (продолжение)

Имя элемента	Тип	Описание
status_validity	int	В случае ошибки это значение указывает, где возникла ошибка - на уровне SCSI или адаптера. Может совпадать с одним из следующих значений встроенной константы: SC_SCSI_ERROR или SC_ADAPTER_ERROR. Если ошибок нет, то значение нулевое.
scsi_status	int	Если поле status_validity равно SC_SCSI_ERROR, то данное поле содержит дополнительную информацию об ошибке. Может совпадать с одним из значений встроенной константы: • SC_GOOD_STATUS (Задача успешно выполнена) • SC_CHECK_CONDITION (Ошибка, дополнительная информация - в данных об ошибке) • SC_BUSY_STATUS (LUN занят, команды не принимаются) • SC_RESERVATION_CONFLICT (Нарушение существующего резервирования SCSI.) • SC_COMMAND_TERMINATED (Устройство завершило команду.) • SC_QUEUE_FULL (Очередь устройства полная.) • SC_ACA_ACTIVE (Устройство находится в состоянии автоматической условной принадлежности.) • SC_TASK_ABORTED (Устройство остановило выполнение команды.) **Примечание:** Определены не все возможные значения. Поэтому SC_SCSI_ERROR может содержать значение, не совпадающее ни с одним из встроенных значений. Можно найти соответствующий код ответа команды SCSI.

Таблица 47. Элементы встроенной переменной __diskcmd (продолжение)

Имя элемента	Тип	Описание
adapter_status	int	Если поле status_validity равно SC_ADAPTER_ERROR, то данное поле содержит дополнительную информацию об ошибке. Может совпадать с одним из следующих значений встроенной константы: • ADAP_HOST_IO_BUS_ERR (ошибка шины ввода-вывода хоста) • ADAP_TRANSPORT_FAULT (ошибка транспортного уровня) • ADAP_CMD_TIMEOUT (истек тайм-аут команды ввода-вывода) • ADAP_NO_DEVICE_RESPONSE (нет ответа от устройства) • ADAP_HDW_FAILURE (аппаратный сбой адаптера) • ADAP_SFW_FAILURE (сбой микрокода адаптера) • ADAP_TRANSPORT_RESET (адаптер обнаружил внешний сброс шины SCSI) • ADAP_TRANSPORT_BUSY (транспортный уровень занят) • ADAP_TRANSPORT_DEAD (транспортный уровень не работает) • ADAP_TRANSPORT_MIGRATED (транспортный уровень перемещен) • ADAP_FUSE_OR_TERMINAL_PWR (перегорел предохранитель адаптера или неисправное электрическое окончание)

Встроенная переменная **__iopath**

Можно использовать специальную встроенную переменную `__iopath` для получения различной информации о пути ввода-вывода для текущей операции. Она доступна в тестах подтипа disk (только для событий `iostart` и `iodone`). К ее элементам можно обращаться с помощью следующего синтаксиса: `__iopath->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как `Invalid Value` (недопустимое значение). Это значение может возвращаться по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра **num_pagefaults** команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

`__iopath` состоит из следующих элементов:

Таблица 48. Элементы встроенной переменной __iopath

Имя элемента	Тип	Описание	Недопустимое значение
path_id	int	ИД текущего пути (начиная с 0).	-1
scsi_id	unsigned long long	ИД SCSI целевого устройства на этом пути.	0xFFFFFFFFFFFFFFFF
lun_id	unsigned long long	Номер логического накопителя (LUN) на этом пути.	0xFFFFFFFFFFFFFFFF
ww_name	unsigned long long	Глобальное имя целевого порта на этом пути.	0
cmds_out	int	Число команд ввода-вывода на этом пути, ожидающих обработки.	-1

Встроенная переменная **__j2info**

Переменная `__j2info` - это встроенная переменная, позволяющая получить различную информацию об операции файловой системы JFS2. Она доступна в тестах подтипа jfs2 (виртуальная файловая система). К ее элементам можно обращаться с помощью следующего синтаксиса: `__j2info->элемент`.

Примечание: Когда фактическое значение получить невозможно, возвращается значение, помеченное как `Invalid Value` (недопустимое значение). Это значение может возвращаться по следующим причинам:

- Требуется контекст страничной ошибки, но текущее значение параметра `num_pagefaults` команды `probevctrl` нулевое или недостаточное.
- Область памяти, содержащая значение, выгружена.
- Любая другая серьезная ошибка системы, например недопустимый указатель или повреждение памяти.

`__j2info` состоит из следующих элементов:

Таблица 49. Элементы встроенной переменной __j2info

Имя элемента	Тип	Описание	Недопустимое значение
inode_id	unsigned long long	Общесистемный уникальный номер, связанный с файлом текущей операции. **Примечание:** Это не номер I-узла файла.	0
f_type	int	Тип файла. Возможные значения приведены в описании `__file->f_type`.	-1
mount_path	char *	Путь к точке монтирования файловой системы.	пустая строка.
devnum	unsigned long long	Номер устройства, на котором расположена файловая система. Он состоит из основного номера и дополнительного номера.	0
major_num	int	Основной номер устройства, на котором расположена файловая система.	-1
minor_num	int	Дополнительный номер устройства, на котором расположена файловая система.	-1
l_blknum	unsigned long long	Номер логического блока для этой файловой операции.	0xFFFFFFFFFFFFFFFF
l_bcount	unsigned long long	Запрошенное количество байтов между логическими блоками в этой операции.	0xFFFFFFFFFFFFFFFF

Таблица 49. Элементы встроенной переменной __j2info (продолжение)

Имя элемента	Тип	Описание	Недопустимое значение
child_bufid	unsigned long long	bufid буфера запроса ввода-вывода, который передается на нижний уровень (например, LVM). На том уровне это значение находится в `__iobuf->bufid`.	0
child_blknum	unsigned long long	Номер блока буфера запроса ввода-вывода, который передается на нижний уровень (например, LVM). На том уровне это значение находится в `__iobuf->blknum`.	0xFFFFFFFFFFFFFFFF
child_bcount	unsigned long long	Количество байтов буфера запроса ввода-вывода, который передается на нижний уровень (например, LVM). На том уровне это значение находится в `__iobuf->bcount`.	0xFFFFFFFFFFFFFFFF
child_bflags	unsigned long long	Флаги буфера запроса ввода-вывода, который передается на нижний уровень (например, LVM). На том уровне это значение находится в `__iobuf->bflags`.	0

Примеры сценариев для администратора тестов ввода-вывода

1. Сценарий трассировки всех операций записи в файл `/etc/passwd`:

```
int write(int, char *, int);
@@BEGIN  {
        target_inodeid = fpath_inodeid("/etc/passwd");
}
@@syscall:*:write:entry {
        if (fd_inodeid(__arg1) == target_inodeid) {
                printf("запись в файл /etc/passwd: timestamp=%A, pid=%lld, pname=[%s], uid=%lld\n",
                        timestamp(), __pid, __pname, __uid);
        }
}
Если сценарии находятся в файле VUE с именем etc_passwd.e. Сценарий можно выполнить следующим образом:
# probevue etc_passwd.e
В другом терминале, если выполняет пользователь (root):
# mkuser user1
Команда probevue выведет следующую информацию:
запись в файл /etc/passwd: timestamp=Mar/03/15 16:10:07, pid=14221508, pname=[mkuser], uid=0
```

2. Сценарий для поиска максимального и минимального времени выполнения операции ввода-вывода для диска, например `hdisk0`, за период времени. Также определяются номер блока, запрошенное количество байтов, время и тип операции (чтение или запись), соответствующей максимальному или минимальному времени.

```
long long min_time, max_time;
@@BEGIN {
        min_time = max_time = 0;
}
@@io:disk:entry:*:hdisk0 {
        ts_entry[__iobuf->bufid] = (long long)timestamp();
}
@@io:disk:exit:*:hdisk0 {
        if (ts_entry[__iobuf->bufid]) { /* только если записано время входа */
                ts_now = timestamp();
                op_type = (__iobuf->bflags & B_READ) ? "ЧТЕНИЕ" : "ЗАПИСЬ";
                dt = (long long)diff_time(ts_entry[__iobuf->bufid], ts_now, MICROSECONDS);
```

```
                if (min_time == 0 || dt < min_time) {
                        min_time = dt;
                        min_blknum = __iobuf->blknum;
                        min_bcount = __iobuf->bcount;
                        min_ts = ts_now;
                        min_optype = op_type;
                }
                if (max_time == 0 || dt > max_time) {
                        max_time = dt;
                        max_blknum = __iobuf->blknum;
                        max_bcount = __iobuf->bcount;
                        max_ts = ts_now;
                        max_optype = op_type;
                }
                ts_entry[__iobuf->bufid] = 0;
        }
}
@@END {
        printf("Максимальное и минимальное время выполнения операции ввода-вывода для [hdisk0]:\n");
        printf("Макс.: %lld мкс, блок=%lld, число байтов=%lld, операция=%s, время операции=[%A]\n",
                max_time, max_blknum, max_bcount, max_optype, max_ts);
        printf("Мин.: %lld мкс, блок=%lld, число байтов=%lld, операция=%s, время операции=[%A]\n",
                min_time, min_blknum, min_bcount, min_optype, min_ts);
}
```

```
Допустим, этот сценарий находится в файле VUE с именем disk_min_max_time.e. Его можно выполнить следующим образом:
# probevue disk_min_max_time.e
Пусть на диске hdisk0 имеет место некоторый ввод-вывод (можно воспользоваться командой dd).
Если завершить вышеупомянутую команду через несколько минут сочетанием клавиш CTRL-C, она выведет следующую информацию:
^CМаксимальное и минимальное время выполнения операций ввода-вывода для [hdisk0]:
Макс.: 48174 мкс, блок=6927976, число байтов=4096, операция=ЧТЕНИЕ, время операции=[Mar/04/15 03:31:07]
Мин.: 133 мкс, блок=6843288, число байтов=4096, операция=ЧТЕНИЕ, время операции=[Mar/04/15 03:31:03]
```

Администратор тестов сети:

Администратор тестов сети отслеживает входящие и исходящие сетевые пакеты в системе (информация пакета интерпретируется модулем bpf в AIX). Спецификация теста позволяет указывать фильтры BPF (пакетный фильтр Berkeley), аналогичные выражениям фильтров программы tcpdump для более точного отслеживания.

Можно использовать встроенные переменные для чтения заголовка и полезной нагрузки пакета для интернет-протоколов. Например, протоколов IPv4, IPv6, TCP, UDP, ICMP, IGMP и ARP.

Администратор тестов сети сообщает о важных событиях протокола (например, изменение состояния TCP, время оборота пакета, повторные передачи, переполнение буфера UDP).

Администратор тестов сети используется главным образом в следующих случаях:

- Предоставление модулю bpf следующей информации о пакете на основе IP-адреса и портов:
  - Отслеживание входящих и исходящих данных для соединения.
  - Для чтения заголовка протокола и полезной нагрузки предусмотрены следующие встроенные переменные.
    - Флаги TCP (SYN, FIN), последовательность TCP и число подтверждения.
    - IPv4/IPv6 (IP-адреса, типы протоколов: tcp, udp, icmp, igmp и т. д.)
    - ICMP (тип пакета: ECHO REQUEST, ECHO RESPONSE и т. д.).
- Обеспечение доступа к полному исходному содержимому сетевого пакета для обработки сценарием теста.
- Информирование о следующих событиях протокола:
  - Отслеживание событий заполнения буфера отправителя и получателя TCP.
  - Переход соединения TCP из состояния SYN-SENT в состояние ESTABLISHED или из состояния ESTABLISHED в состояние CLOSE.

– Мониторинг времени перехода между состояниями (например, сколько времени занял переход из состояния SYN-SENT в состояние ESTABLISHED).
– Обнаружение получателей (информация о соединении), отклоняющих соединения из-за переполнения очереди.
– Обнаружение повторных передач (вторая и последующие повторные передачи пакета) для соединений TCP.
– Обнаружение потери пакетов сокетом UDP из-за слишком маленького буфера приема.

Спецификация теста

Спецификация теста для администратора тестов сети состоит из трех или пяти кортежей, разделенных двоеточием (:). Первый кортеж всегда `@@net`.

Администратор тестов сети поддерживает две основные категории спецификаций: одна предназначена для сбора информации о пакете, другая - для сбора информации о протоколе.

- Формат для сбора информации о пакете:
  `@@net:bpf:<интерфейс-1>|<интерфейс-2>|.....:<протокол>:<фильтр>`
- Формат для сбора информации о протоколе
  `@@net:tcp:<имя-события>`
  `@@net:udp:<имя-события>`

Подтип теста

Второй кортеж обозначает подтип теста, указывающий уровень сетевого стека AIX, содержащий тест. Этот кортеж может иметь одно из следующих значений (* нельзя указывать):

Таблица 50. Спецификация второго кортежа для подтипа теста

Второй кортеж (подтип)	**Описание**
bpf	Этот тест запускается на уровне сетевого интерфейса, когда пакет соответствует определенному фильтру.
tcp	Этот тест запускается для событий протокола TCP.
udp	Этот тест запускается для событий протокола UDP.

Проверка сетевого события или сбор информации о сетевом пакете

Третий кортеж зависит от подтипа (второй кортеж). Его значение не может быть *.

Тесты bpf

Спецификация содержит 5 кортежей для тестов bpf (см. следующую таблицу).

Таблица 51. Тесты bpf. Спецификация кортежей

Второй кортеж (подтип)	Следующие кортежи	Описание
bpf	Третий кортеж - имена интерфейсов	В этом кортеже указывается интерфейс или список интерфейсов, для которых будет собираться информация о пакетах. Возможные значения: `enX` (например, `en0`,`en1`) и `lo0`. Значение * не поддерживается для этого кортежа. Можно указать один или несколько интерфейсов, используя символ | в качестве разделителя.
	Четвертый кортеж - протокол	В этом кортеже указывается протокол для запуска теста. Возможные значения: `ether`, `arp`, `rarp`, `ipv4`, `ipv6`, `tcp`, `udp`, `icmp4`, `icmp6` и `igmp`. Заполняются встроенные переменные для протокола, доступные из сценария Vue. Например, значение протокола ipv4 заполняет встроенные переменные `__ip4hdr`. Значение * для этого кортежа указывает, что тест запускается для всех типов протоколов, соответствующих указанному фильтру. Когда протокол - *, встроенные переменные, поддерживаемые администратором тестов сети, недоступны в сценариях Vue. Исходные данные пакета запрошенного размера можно получить с помощью функции Vue `copy_kdata()` и отобразить на заголовки соответствующих протоколов. **Примечание:** Значение * может ухудшить производительность, поскольку тест будет запускаться для всех входящих и исходящих пакетов на указанных интерфейсах, которые соответствуют фильтру. Также имеет место копирование, когда информация пакета занимает несколько буферов пакетов.
	Пятый кортеж - строка фильтра bpf	В этом кортеже указывается выражение фильтра bpf (выражения фильтра имеют такой же синтаксис, как в команде **tcpdump**). Выражение фильтра должно указываться в двойных кавычках. Выражение фильтра и протокол, указанный в четвертом кортеже, должны быть совместимыми. Значение * не поддерживается в этом кортеже. Дополнительную информацию о выражениях фильтра можно найти в документации **tcpdump**.

Примеры

1. Формат спецификации для доступа к встроенным переменным, связанным с заголовком Ethernet (`__etherhdr`), заголовком IP (`__ip4hdr` или `__ip6hdr`) и заголовком TCP (`__tcphdr`), из сценария Vue, когда интерфейс en0 принимает или отправляет пакеты через порт 23 (строка фильтра "port 23"):

   ```
   @@net:bpf:en0:tcp:"port 23"
   ```

2. Формат спецификации для доступа ко встроенным переменным, связанным с заголовком Ethernet (`__etherhdr`), IP-заголовком (`__ip4hdr` или `__ip6hdr`) и заголовком UDP (`__udphdr`) из сценария Vue, если система взаимодействует с хостом example.com (строка фильтра “example.com”) через интерфейсы en0 и en1:

   ```
   @@net:bpf:en0|en1:udp:"host example.com"
   ```

3. Формат спецификации для доступа к исходным данным пакета, когда система принимает или отправляет пакеты для "host example.com":

   ```
   @@net:bpf:en1:*:"host example.com"
   ```

Примечание: Каждая спецификация теста bpf использует устройство bpf. Эти устройства являются общими для ProbeVue, **tcpdump** и другими программами, которые используют библиотеку libpcap или bpf для захвата и внедрения пакетов. Число тестов bpf зависит от числа доступных устройств bpf в системе.

При запуске теста bpf исходные данные пакета помещаются в переменную *__mdata*. Исходные данные пакета запрошенного размера можно получить с помощью функции Vue copy_kdata () и отобразить их на структуру ether_header, ip header и т. п. Следующие структуры применяются для извлечения заголовка и полезной нагрузки.

Пример

```
Сценарий VUE для доступа к исходным данным пакета, когда для протокола указано значение "*".

 /* Определение структуры заголовка ether */
struct  ether_header {
        char  ether_dhost[6];
        char  ether_shost[6];
        short ether_type;
};

/* Сценарий ProbeVue для доступа к исходным данным пакета и их интерпретации */

@@net:bpf:en0:*:"port 23"
{
        /* определение локальных переменных сценария */
        __auto struct ether_header eth;
        __auto char *mb;

        /* __mdata содержит адрес данных пакета */
        mb =(char *) __mdata;
        printf("probevue для сети\n");

       /*
        * Применение функции VUE copy_kdata() для копирования данных
        * запрошенного размера (размер структуры ether_header) из mbuf в переменную eth
        * (тип ether_header).
        */
        copy_kdata (mb, eth);
        printf("Тип из исходных данных: %x\n",eth.ether_type);

}
```

Тесты TCP

Спецификация содержит три кортежа для тестов TCP (см. следующую таблицу).

Таблица 52. Тесты TCP. Спецификация кортежей

Второй кортеж (подтип)	События (третий кортеж) Значение * не поддерживается в этом кортеже.	Описание
tcp	state_change	Этот тест запускается при изменении состояния TCP.
	send_buf_full	Этот тест запускается при возникновении события заполнения буфера отправки.
	recv_buf_full	Этот тест запускается при возникновении события заполнения буфера приема.
	retransmit	Этот тест запускается при повторной передаче пакета через соединение TCP.
	listen_q_full	Этот тест запускается, когда сервер (сокет получателя запросов) отклоняет новые запросы на установку соединения из-за отсутствия места в очереди получателя.

Во встроенную переменную **__proto_info** помещается информация о соединении TCP (локальный IP-адрес, удаленный IP-адрес, локальный порт и удаленный порт) при возникновении события TCP. Удаленный порт и удаленный IP-адрес имеют значение NULL для события listen_q_full.

Пример

Спецификации тестов для изменений состояния протокола TCP:

```
@@net:tcp:state_change
```

Тесты udp

Для тестов `udp` спецификация состоит из трех кортежей (см. следующую таблицу).

Таблица 53. Значения третьего кортежа, когда второй кортеж - udp

Второй кортеж (подтип)	События (третий кортеж) Значение * не поддерживается в этом кортеже.	Описание
udp	sock_recv_buf_overflow	Этот тест запускается, когда возникает переполнение дейтаграммы или буфера приема данных сокета UDP.

Встроенная переменная `__proto_info` позволяет получить данные протокола UDP (исходный и целевой IP-адреса и номера портов), если возникает событие переполнения буфера приема.

```
@@net:udp:sock_recv_buf_overflow
```

Пример

Спецификации тестов для переполнения буфера приема данных сокета UDP:

```
@@net:udp:sock_recv_buf_overflow
```

Встроенные переменные для сценариев Vue, связанные с тестами сети

Сетевые события можно тестировать с помощью следующих встроенных переменных.

Встроенная переменная **__etherhdr**

Переменная __etherhdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке ether из отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с одним из следующих протоколов: ipv4, tcp, udp, icmp4 и rarp. Эта переменная доступна в тестах подтипа `bpf`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__etherhdr->элемент`.

Значение встроенной переменной __etherhdr состоит из следующих элементов:

Таблица 54. Элементы встроенной переменной __etherhdr

Имя элемента	Тип	Описание
src_addr	mac_addr_t	Исходный MAC-адрес. Тип данных mac_addr_t используется для хранения MAC-адреса. MAC-адрес можно вывести с помощью спецификатора формата "M".
dst_addr	mac_addr_t	Целевой MAC-адрес. Тип данных mac_addr_t используется для хранения MAC-адреса. MAC-адрес можно вывести с помощью спецификатора формата "M".
ether_type	unsigned short	Это имя указывает на протокол, инкапсулированный в полезной нагрузке кадра Ethernet. Возможные протоколы: IPv4, IPv6, ARP и REVARP. Может совпадать с одним из следующих значений встроенной константы для ether_type: • ETHERTYPE_IP • ETHERTYPE_IPV6 • ETHERTYPE_ARP • ETHERTYPE_REVARP См. значения ether_type в файлах заголовков `/usr/include/netinet/if_ether.h` и `/usr/include/netinet/if_ether6.h`.

Примечание: Встроенная переменная __etherhdr применяется только для интерфейсов Ethernet, для циклических интерфейсов она недоступна.

Встроенная переменная __ip4hdr

Переменная __ip4hdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке IPv4 из отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с одним из следующих протоколов: ipv4, tcp, udp, icmp4 и igmp. И содержит допустимые данные, когда версия протокола IP - IPv4. Эта переменная доступна в тестах подтипа `bpf`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__ip4hdr->элемент`.

Эта встроенная переменная содержит следующие элементы:

Таблица 55. Элементы встроенной переменной __ip4hdr

Имя элемента	Тип	Описание
src_addr	ip_addr_t	Исходный IP-адрес. Тип данных ip_addr_t используется для хранения IP-адреса. Для вывода IP-адреса в десятичном формате с точками используется спецификатор формата "I", для вывода имени хоста используется спецификатор формата "H". Вывод имени хоста - дорогостоящая операция.
dst_addr	ip_addr_t	Целевой IP-адрес. Тип данных ip_addr_t используется для хранения IP-адреса. Для вывода IP-адреса в десятичном формате с точками используется спецификатор формата "I", для вывода имени хоста используется спецификатор формата "H". Вывод имени хоста - дорогостоящая операция.

protocol	unsigned short	Этот элемент указывает протокол раздела данных дейтаграммы IP. Возможные протоколы: TCP, UDP, ICMP, IGMP, FRAGMENTED и пр.
		
		Может совпадать с одним из следующих значений встроенной константы для протокола.
		`IPPROTO_HOPOPTS,`
		`IPPROTO_ICMP,`
		`IPPROTO_IGMP,`
		`IPPROTO_TCP,`
		`IPPROTO_UDP,`
		`IPPROTO_ROUTING,`
		`IPPROTO_FRAGMENT,`
		`IPPROTO_NONE,`
		`IPPROTO_LOCAL`
		
		Значения протоколов определены в заголовочном файле `/usr/include/netinet/in.h`.

ttl	unsigned short	Время хранения в кэше или ограничение на пересылку.
cksum	unsigned short	Контрольная сумма заголовка IP.
id	unsigned short	Идентификационный номер. Этот элемент используется в качестве уникального идентификатора группы фрагментов одной дейтаграммы IP.
total_len	unsigned short	Суммарная длина. Это значение - полный размер пакета (фрагмента), включая заголовок IP и данные, в байтах.
hdr_len	unsigned short	Размер заголовка IP.
tos	unsigned short	Тип службы.

Таблица 55. Элементы встроенной переменной __ip4hdr (продолжение)

Имя элемента	Тип	Описание
frag_offset	unsigned short	Смещение фрагмента. Это значение - смещение определенного фрагмента, относительно начала исходной, нефрагментированной, дейтаграммы IP. Первый фрагмент имеет нулевое смещение. Может совпадать с одним из значений флага встроенной константы `frag_offset`. Значения флагов должны быть побитовыми и содержать значение флага встроенной константы для проверки наличия конкретного флага: • IP_DF (флаг отсутствия фрагмента) • IP_MF (флаг дополнительных фрагментов) Значения флагов определены в заголовочном файле `/usr/include/netinet/ip.h`.

Встроенная переменная __ip6hdr

Переменная __ip6hdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке IPv6 из отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса. Эта переменная с любым из протоколов ipv6, tcp, udp и icmp6 содержит допустимые данные, когда версия протокола IP - IPv6. Эта переменная доступна в тестах подтипа `bpf`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__ip6hdr->элемент`.

Эта встроенная переменная содержит следующие элементы:

Таблица 56. Элементы встроенной переменной __ip6hdr

Имя элемента	Тип	Описание
src_addr	ip_addr_t	Исходный IP-адрес. Тип данных ip_addr_t используется для хранения IP-адреса. Для вывода IP-адреса используется спецификатор формата "I", для вывода имени хоста используется спецификатор формата "H". Вывод имени хоста - дорогостоящая операция.
dst_addr	ip_addr_t	Целевой IP-адрес. Тип данных ip_addr_t используется для хранения IP-адреса. Для вывода IP-адреса используется спецификатор формата "I", для вывода имени хоста используется спецификатор формата "H". Вывод имени хоста - дорогостоящая операция.

Таблица 56. Элементы встроенной переменной __ip6hdr (продолжение)

Имя элемента	Тип	Описание
protocol	unsigned short	Это значение указывает протокол раздела данных дейтаграммы IP. Возможные протоколы: TCP, UDP, ICMPv6 и пр. Может совпадать с одним из следующих значений встроенной константы для протокола:`IPPROTO_TCP`,`IPPROTO_UDP`, `IPPROTO_ROUTING`, `IPPROTO_ICMPV6`, `IPPROTO_NONE`, `IPPROTO_DSTOPTS`, `IPPROTO_LOCAL` Значения протоколов определены в заголовочном файле `/usr/include/netinet/in.h`.
hop_limit	unsigned short	Ограничение на пересылку (время хранения в кэше).
total_len	unsigned short	Суммарная длина (размер полезной нагрузки). Размер полезной нагрузки, включая заголовки расширений.
next_hdr	unsigned short	Тип следующего заголовка. Это поле обычно указывает на протокол транспортного уровня, который используется полезной нагрузкой пакета. Когда в пакете есть заголовки расширений, это поле указывает на следующий заголовок расширения. Значения этого поля такие же, как у соответствующего поля протокола IPv4.
flow_label	unsigned int	Метка потока.
traffic_class	unsigned int	Класс потока данных.

Встроенная переменная __tcphdr

Переменная __tcphdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке tcp из отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом TCP. Она доступна в тестах подтипа bpf. К ее элементам можно обращаться с помощью следующего синтаксиса: `__tcphdr->элемент`.

Встроенная переменная __tcphdr состоит из следующих элементов:

Таблица 57. Элементы встроенной переменной __tcphdr

Имя элемента	Тип	Описание
src_port	unsigned short	Исходный порт пакета.
dst_port	unsigned short	Целевой порт пакета.

Таблица 57. Элементы встроенной переменной __tcphdr (продолжение)

Имя элемента	Тип	Описание
flags	unsigned short	Биты, указывающие на передачу управляющей информации. По 1 биту на флаг. Может совпадать с одним из значений флага встроенной константы. Значения флагов должны быть побитовыми и содержать значение флага встроенной константы для проверки наличия конкретного флага. • TH_FIN (Больше нет данных от отправителя) • TH_SYN (Запрос на установку соединения) • TH_RST (Сброс соединения) • TH_PUSH (Функция Push. Запрашивает передачу буферизованных данных приложению-получателю) • TH_ACK (Указывает, что данный пакет содержит подтверждение) • TH_URG (Указывает, что поле индикатора срочности значимое) Дополнительную информацию об этих флагах можно найти в документации протокола TCP, значения флагов определены в заголовочном файле `/usr/include/netinet/tcp.h`.
seq_num	unsigned int	Порядковый номер.
ack_num	unsigned int	Число подтверждения.
hdr_len	unsigned int	Длина заголовка TCP
cksum	unsigned short	Контрольная сумма.
window	unsigned short	Размер окна.
urg_ptr	unsigned short	Индикатор срочности.

Встроенная переменная __udphdr

Переменная __udphdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке `udp` из отфильтрованного пакета. Эта встроенная переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом udp. Она доступна в тестах подтипа `bpf`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__udphdr->элемент`.

Встроенная переменная __udphdr содержит следующие элементы:

Таблица 58. Элементы встроенной переменной __udphdr

Имя элемента	Тип	Описание
src_port	unsigned short	Исходный порт пакета.
dst_port	unsigned short	Целевой порт пакета.
length	unsigned short	Длина заголовка и данных UDP.
cksum	unsigned short	Контрольная сумма.

Встроенная переменная __icmp

Переменная __icmp - это специальная встроенная переменная, позволяющая получить информацию о заголовке icmp из отфильтрованного пакета. Эта встроенная переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом icmp. Она доступна в тестах подтипа bpf. К ее элементам можно обращаться с помощью следующего синтаксиса: `__icmp->элемент`.

Эта встроенная переменная содержит следующие элементы:

Таблица 59. Элементы встроенной переменной __icmp

Имя элемента	Тип	Описание

type	unsigned short	Тип сообщения ICMP.
		
		Пример: 0 - эхо-ответ, 8 - эхо-запрос, 3 - целевой адрес недоступен. Есть и другие типы. См. документацию на сетевые стандарты.
		
		Может совпадать с одним из следующих значений встроенной константы для типов сообщений ICMP:
		`ICMP_ECHOREPLY,`
		`ICMP_UNREACH`
		`ICMP_SOURCEQUENCH,`
		`ICMP_REDIRECT,`
		`ICMP_ECHO,`
		`ICMP_TIMXCEED,`
		`ICMP_PARAMPROB,`
		`ICMP_TSTAMP,`
		`ICMP_TSTAMPREPLY,`
		`ICMP_IREQ,`
		`ICMP_IREQREPLY,`
		`ICMP_MASKREQ,`
		`ICMP_MASKREPLY`
		
		Значения протокола определены в заголовочном файле `/usr/include/netinet/ip_icmp.h`.
		Примечание: Определены не все возможные значения типов сообщений, поэтому значение может быть и другим.

Таблица 59. Элементы встроенной переменной __icmp (продолжение)

Имя элемента	Тип	Описание

code	unsigned short	Подтип сообщения ICMP.
		
		Для каждого типа сообщений определено несколько разных кодов и подтипов. Например, нет маршрута к целевому адресу, связь с целевым адресом административно запрещена, нет соседнего узла, адрес недоступен, порт недоступен. См. документацию на сетевые стандарты.
		
		Может совпадать с одним из следующих значений встроенной константы для подтипов ICMP:
		
		ICMP_UNREACH_NET ICMP_UNREACH_HOST ICMP_UNREACH_PROTOCOL ICMP_UNREACH_PORT ICMP_UNREACH_NEEDFRAG ICMP_UNREACH_SRCFAIL ICMP_UNREACH_NET_ADMIN_PROHIBITED ICMP_UNREACH_HOST_ADMIN_PROHIBITED
		
		Значения подтипов для типа 4
		
		Значения подтипов для типа 4 следующие:
		`ICMP_REDIRECT_NET`
		`ICMP_REDIRECT_HOST`
		`ICMP_REDIRECT_TOSNET`
		`ICMP_REDIRECT_TOSHOST`
		
		Значения подтипов для типа 6
		
		Значения подтипов для типа 6 следующие:
		`ICMP_TIMXCEED_INTRANS`
		`ICMP_TIMXCEED_REASS`
		
		Значения подтипов для типа 7
		
		Значения подтипов для типа 7 следующие:
		
		ICMP_PARAMPROB_PTR
		
		ICMP_PARAMPROB_MISSING
		
		Значения подтипов сообщений определены в заголовочном файле /usr/include/netinet/ip_icmp.h.
		
		Примечание: определены не все возможные подтипы сообщений, поэтому значение подтипа сообщения может быть и другим.

cksum	unsigned short	Контрольная сумма.

Встроенная переменная __icmp6

Переменная __icmp6 - это специальная встроенная переменная, содержащая заголовок icmpv6 отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом `icmp6`. Она доступна в тестах подтипа `bpf`. Элементы этой встроенной переменной доступны посредством синтаксиса "__icmp6->элемент".

__icmp6 состоит из следующих элементов:

Таблица 60. Элементы встроенной переменной __icmp6

Имя элемента	Тип	Описание

type	unsigned short	Тип сообщения ICMPV6.
		
		Это тип сообщения, определяющий формат остальных данных.
		
		Может совпадать с одним из следующих значений встроенной константы для типов ICMPV6.
		`ICMP6_DST_UNREACH`
		`ICMP6_PACKET_TOO_BIG`
		`ICMP6_TIME_EXCEEDED`
		`ICMP6_PARAM_PROB`
		`ICMP6_INFOMSG_MASK`
		`ICMP6_ECHO_REQUEST`
		`ICMP6_ECHO_REPLY`
		
		Значения протокола определены в заголовочном файле `/usr/include/netinet/icmp6.h`.
		Примечание: Определены не все возможные значения типов сообщений, поэтому значение может быть и другим.

code	unsigned short	Подтип сообщения ICMPV6.
		
		Это значение зависит от типа сообщения. Оно дает дополнительный уровень дискретности сообщения.
		
		Может совпадать с одним из следующих значений встроенной константы для подтипов ICMPV6.
		`ICMP6_DST_UNREACH_NOROUTE`
		`ICMP6_DST_UNREACH_ADMIN`
		`ICMP6_DST_UNREACH_ADDR`
		`ICMP6_DST_UNREACH_BEYONDSCOPE`
		`ICMP6_DST_UNREACH_NOPORT`
		
		Значения подтипов сообщений определены в заголовочном файле `/usr/include/netinet/icmp6.h`.
		Примечание: Определены не все возможные значения подтипов сообщений, поэтому значение может быть и другим.

cksum	unsigned short	Контрольная сумма.

Встроенная переменная __igmp

__igmp - это специальная встроенная переменная, содержащая заголовок `igmp` отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом `igmp`. Эта переменная доступна в тестах подтипа `bpf`. Ее элементы доступны посредством синтаксиса "__igmp->элемент".

Встроенная переменная __igmp содержит следующие элементы:

Таблица 61. Элементы встроенной переменной __igmp

Имя элемента	Тип	Описание

type	unsigned short	Тип сообщения IGMP.
		
		Пример: `Запрос членства (0x11)`, `Отчет о членстве (IGMPv1: 0x12, IGMPv2: 0x16, IGMPv3: 0x22)`, `Покинуть группу (0x17)` См. документацию стандарта или документацию по сетям.
		
		Может совпадать с одним из следующих значений встроенной константы для типов сообщений IGMP.
		`IGMP_HOST_MEMBERSHIP_QUERY`
		`IGMP_HOST_MEMBERSHIP_REPORT`
		`IGMP_DVMRP`
		`IGMP_HOST_NEW_MEMBERSHIP_REPORT`
		`IGMP_HOST_LEAVE_MESSAGE`
		`IGMP_HOST_V3_MEMBERSHIP_REPORT`
		`IGMP_MTRACE`
		`IGMP_MTRACE_RESP`
		`IGMP_MAX_HOST_REPORT_DELAY`
		
		Значения протоколов определены в заголовочном файле `/usr/include/netinet/igmp.h`.
		
		Примечание: определены не все возможные значения типов сообщений, поэтому значение может быть и другим.

code	unsigned short	Подтип типа IGMP.
		
		Может совпадать с одним из следующих значений встроенной константы для подтипов сообщений IGMP.
		
		Значения подтипов для типа 3.
		`DVMPP_PROBE 1`
		`DVMRP_REPORT 2`
		`DVMRP_ASK_NEIGHBORS 3`
		`DVMRP_ASK_NEIGHBORS2 4`
		`DVMRP_NEIGHBORS 5`
		`DVMRP_NEIGHBORS2 6`
		`DVMRP_PRUNE 7`
		`DVMRP_GRAFT 8`
		`DVMRP_GRAFT_ACK 9`
		`DVMRP_INFO_REQUEST 10`
		`DVMRP_INFO_REPLY 11`
		
		Примечание: Определены не все возможные значения подтипов сообщений, поэтому значение может быть и другим.

cksum	unsigned short	Значение контрольной суммы IGMP.
group_addr	ip_addr_t	Групповой адрес, сообщаемый или запрашиваемый. Этот адрес представляет собой групповой адрес, который запрашивается при отправке запроса, связанного с группой или с группой и источником. Поле имеет нулевое значение, если отправляется общий запрос. Тип данных ip_addr_t используется для хранения IP-адреса группы. IP-адрес можно вывести с помощью спецификатора формата "I".

Встроенная переменная **__arphdr**

Переменная __arphdr - это специальная встроенная переменная, позволяющая получить информацию о заголовке `arphdr` из отфильтрованного пакета. Эта переменная доступна в тестах пакетной информации на уровне интерфейса с протоколом `arp` или `rarp`. Она доступна в тестах подтипа `bpf`. К элементам переменной __arphdr можно обращаться посредством синтаксиса `__arphdr->элемент`.

Встроенная переменная __arphdr состоит из следующих элементов:

Таблица 62. Элементы встроенной переменной __arphdr

Имя элемента	Тип	Описание

hw_addr_type	unsigned short	Формат аппаратного типа адреса. Это поле содержит конкретный протокол канала передачи данных, который используется.
		
		Может совпадать с одним из следующих значений встроенной константы для протокола передачи данных:
		`ARPHRD_ETHER,`
		`ARPHRD_802_5,`
		`ARPHRD_802_3 и`
		`ARPHRD_FDDI`
		
		Значения протокола определены в заголовочном файле `/usr/include/net/if_arp.h`.

protocol_type	unsigned short	Формат типа адреса протокола. Это поле содержит конкретный сетевой протокол, который используется.
		
		Может совпадать с одним из следующих значений встроенной константы для сетевого протокола:
		`SNAP_TYPE_IP,`
		`SNAP_TYPE_AP,`
		`SNAP_TYPE_ARP,`
		`VLAN_TAG_TYPE`
		
		Значения протокола определены в заголовочном файле `/usr/include/net/nd_lan.h`.

hdr_len	unsigned short	Длина аппаратного адреса или MAC-адреса.
proto_len	unsigned short	Длина адреса протокола или IP-адреса.

operation	unsigned short	Операция, выполняемая отправителем: 1 - запрос, 2 - ответ.
		
		Может совпадать с одним из следующих значений встроенной константы для сетевого протокола:
		`ARPOP_REQUEST,`
		`ARPOP_REPLY`
		
		Значения протокола определены в заголовочном файле `/usr/include/net/if_arp.h`.

Таблица 62. Элементы встроенной переменной __arphdr (продолжение)

Имя элемента	Тип	Описание
src_mac_addr	mac_addr_t	MAC-адрес отправителя или исходный MAC-адрес. Аппаратный адрес или MAC-адрес отправителя хранится в типе данных mac_addr_t. Для вывода аппаратного адреса или MAC-адреса отправителя используется спецификатор формата "%M".
dst_mac_addr	mac_addr_t	Целевой MAC-адрес. Целевой аппаратный адрес или MAC-адрес хранится в типе данных mac_addr_t. Для вывода целевого аппаратного адреса или MAC-адреса используется спецификатор формата "%M".
src_ip	ip_addr_t	Исходный IP-адрес или IP-адрес отправителя. Исходный IP-адрес хранится в типе данных `ip_addr_t`. Для вывода IP-адреса отправителя используется спецификатор формата "%I".
dst_ip	ip_addr_t	Целевой IP-адрес. Целевой IP-адрес хранится в типе данных `ip_addr_t`. Для вывода целевого IP-адреса используется спецификатор формата "%I".

Пример

Сценарий Vue для тестирования информации заголовков пакетов, принимаемых или передаваемых через порт 23. Выводит информацию об исходном и целевом узлах, а также длину заголовка `tcp`

```
@@net:bpf:en0:tcp:"port 23"
{
        printf("src_addr:%I b dst_addr:%I\n",__ip4hdr->src_addr,__ip4hdr->dst_addr);
        printf("src port:%d\n",__tcphdr->src_port);
        printf("dst port:%d\n",__tcphdr->dst_port);
        printf("tcp hdr_len:%d\n",__tcphdr->hdr_len);
}

Вывод:
# probevue bpf_tcp.e
src_addr:10.10.10.12 и dst_addr:10.10.18.231
src port:48401
dst port:23
tcp hdr_len:20
..................
..................
```

Встроенная переменная **__proto_info**

Переменная __proto_info - это специальная встроенная переменная, позволяющая получить информацию о протоколе (исходные и целевые IP-адреса и порты) для событий TCP и UDP. Переменная __proto_info доступна в тестах подтипа `tcp` или `udp`. К ее элементам можно обращаться с помощью следующего синтаксиса: `__proto_info->элемент`.

Встроенная переменная __proto_info состоит из следующих элементов:

Таблица 63. Элементы встроенной переменной __proto_info

Имя элемента	Тип	Описание
local_port	unsigned short	Локальный порт
remote_port	unsigned short	Удаленный порт
local_addr	ip_addr_t	Локальный адрес
remote_addr	ip_addr_t	Удаленный адрес

Дополнительная информация для событий TCP

В следующей таблице описываются события изменения состояния TCP:

Таблица 64. События изменения состояния TCP

Имя	Тип	Описание
__prev_state	short	Информация о предыдущем состоянии для соединения.
__cur_state	short	Информация о текущем состоянии для соединения.
		Может совпадать с одним из следующих значений встроенной константы для состояний TCP: • TCPS_ESTABLISHED (соединение установлено) • TPCS_CLOSED (соединение закрыто) • TPCS_LISTEN (прием запросов на установку соединения) • TPCS_SYN_SENT (отправлен пакет SYN удаленному узлу) • TCPS_SYN_RECEIVED (принят пакет SYN от удаленного узла) • TCPS_CLOSE_WAIT (получен пакет FIN, ожидание закрытия) • TCPS_FIN_WAIT_1 (закрыто, отправлен пакет FIN) • TCPS_CLOSING (закрыто, выполнен обмен пакетами FIN, ожидание пакета FIN ACK) • TCPS_LAST_ACK (получены пакет FIN и пакет закрытия, ожидание пакета FIN ACK) • TCPS_FIN_WAIT_2 (закрыто, пакет FIN подтвержден) • TCPS_TIME_WAIT (ожидание в течение 2*msl после закрытия) Значения определены в экспортированном заголовочном файле `/usr/include/netinet/tcp_fsm.h`.

Например:

Следующий сценарий Vue выводит информацию об изменении состояния для определенного соединения:

```
@@net:tcp:state_change
when(__proto_info->local_addr =="10.10.10.1" and __proto_info->remote_addr == 10.10.10.2"
 and __proto_info->local_port == "8000" and __proto_info->remote_port == "9000")
{
 printf("Предыдущее состояние: %d, текущее состояние: %d\n",__prev_state,__cur_state);
}
```

Событие повторной передачи TCP

Таблица 65. Событие повторной передачи TCP

Имя	Тип	Описание
__nth_retransmit	unsigned short	n-я повторная передача

Примеры

1. В следующем примере обнаруживается получатель, отклоняющий соединения из-за переполнения очереди.

```
@@net:tcp:listen_q_full
{
 printf("IP-адрес получателя: %I и номер порта: %d\n",__proto_info->local_addr, __proto_info->local_port);
}
```

2. В следующем примере обнаруживается соединение, теряющее пакеты из-за переполнения буфера сокета

```
@@net:udp:sock_recv_buf_overflow
{
        printf("Информация о соединении, теряющем пакеты из-за переполнения буфера сокета:\n");
        printf("Локальный IP-адрес: %I и удаленный IP-адрес: %I\n",__proto_info->local_addr,__proto_info->remote_addr);
        printf("локальный порт: %d и удаленный порт: %d\n",__proto_info->local_port, __proto_info->remote_port);
 }
```

3. Обнаружение повторных передач (вторая и последующие повторные передачи пакета) для определенного соединения TCP.

```
@@net:tcp:retransmit
when (__proto_info->local_addr == "10.10.10.1" &&
  __proto_info->remote_addr == "10.10.10.2" &&
__proto_info->local_port == "4000" &&
  __proto_info->remote_port == "5000")
{
        printf(" %d th re-transmition for this connection\n", _nth_retransmit);
}
```

4. Получение информации о соединении при возникновении события заполнения буфера отправителя.

```
@@net:tcp:send_buf_full
{
        printf("Информация о соединении при возникновении события заполнения буфера:\n");
        printf("Локальный IP-адрес: %I и удаленный IP-адрес: %I\n",__proto_info->local_addr,__proto_info->remote_addr);
        printf("локальный порт: %d и удаленный порт: %d\n",__proto_info->local_port, __proto_info->remote_port);
 }
```

Администратор тестов sysproc:
Обзор

Администратор тестов sysproc предоставляет пользователям и администраторам инфраструктуру для динамической трассировки процессов и нитей без необходимости изучать внутреннее устройство подсистемы sysproc.

Аспекты подсистемы sysproc для пользователей и администраторов делятся на следующие основные категории:

- Создание и завершение процесса/нити
- Генерация и доставка сигналов
- Планирование и диспетчеризация событий

- События динамического изменения и привязки CPU

Создание и завершение процесса/нити

Информация, связанная с созданием и уничтожением процессов и нитей, требуется системным администраторам для управления ресурсами системы. Администратор тестов sysproc применяется в следующих важных случаях:

- Штатно завершился процесс или из-за ошибки?
- Когда процессы или нити создаются, завершаются или превышают ограничения?
- Сколько выполнялся процесс?
- Отслеживание событий, когда нить получает исключительную ситуацию или возвращается из исключительной ситуации.

Генерация и доставка сигналов

Сигналы влияют на текущее состояние процесса или нити в системе. По состоянию сигналов и текущему состоянию процессов вследствие этих сигналов администратор может установить причину неправильного поведения процессов или нитей. Важные случаи из категории генерации и доставки сигналов, в которых применяется данный администратор тестов, включают следующие (но не ограничиваются ими):

- Источник и информация сигнала для определенного получателя.
- Доставка асинхронных сигналов.
- Трассировка очистки сигналов.
- Трассировка случаев замены обработчика сигналов по умолчанию.
- Получатель и информация сигнала для определенного источника.
- Трассировка входа в обработчик сигнала и выхода из него.

Планирование и диспетчеризация событий

Планировщик и диспетчер управляют выполнением процессов и нитей в системе. Администратор анализирует производительность системы с помощью подсистемы динамической трассировки планировщика или диспетчера.

Подсистема динамической трассировки планировщика или диспетчера помогает понять причины незавершения нитей.

Ниже перечислены некоторые важные случаи из категории событий планировщика и диспетчера, в которых применяется администратор тестов sysproc.

- Трассировка постановки нитей в очередь выполнения и удаления их из очереди выполнения.
- Трассировка событий замещения нитей в системе.
- Трассировка переключения нити в состояние сна по событию.
- Трассировка переключения нити из состояния сна в активное состояние.
- Отслеживание задержек диспетчеризации нити.
- Отслеживание событий свертывания виртуального процессора.
- Трассировка изменений приоритета нитей ядра.

События динамического изменения и привязки CPU

Этот класс тестов дает возможность динамической трассировки пользователям, отслеживающим ресурсы, связанные с процессом.

Некоторые важные случаи из данной категории, в которых применяется администратор тестов событий динамического изменения и привязки CPU:

- Отслеживание изменения привязки нити к CPU.
- Отслеживание подключения ресурсов к процессу и их отключения.
- Отслеживание событий привязки к CPU.
- Отслеживание событий начала или окончания динамического изменения.

Спецификация теста

Для тестирования событий sysproc в сценариях Vue должен использоваться следующий формат:

```
@@sysproc:<событие-sysproc>:<pid/tid/*>
```

Первый кортеж `@@sysproc` указывает, что этот тест предназначен для событий sysproc.

Второй кортеж указывает тестируемое событие.

Третий кортеж играет роль фильтра для изоляции событий, указанных во втором кортеже, по ИД процесса или нити ядра.

Примечание: Использование ИД процесса или нити ядра в качестве фильтра в тестах sysproc не гарантирует, что событие возникнет в контексте процесса или нити. Администратор тестов sysproc использует ИД процесса или нити только как фильтр. Эти события могут быть полезными с точки зрения процесса или нити независимо от контекста выполнения события теста.

Событие отправки сигнала, где процесс либо отправляет сигнал, либо принимает его, может быть полезным. В следующей информации указываются подходящие фильтры для таких событий теста.

Точки тестирования (интересующие события)

В следующей таблице дано краткое описание всех событий, которые можно тестировать с помощью администратора тестов sysproc:

Таблица 66. События теста sysproc

Тест (sysproc_event)	Описание
forkfail	Отслеживание ошибок функции fork.
execfail	Отслеживание ошибок функции exec.
execpass	Отслеживание успешных выполнений функции exec.
exit	Отслеживание выхода из процесса.
threadcreate	Отслеживание создания нитей ядра.
threadterminate	Отслеживание завершения нитей ядра.
threadexcept	Отслеживание исключительных ситуаций процесса.
sendsig	Отслеживание отправки сигнала в процесс внешними источниками.
sigqueue	Отслеживание постановки сигналов в очередь сигналов процесса
sigdispose	Отслеживание обработки сигналов.
sigaction	Отслеживание установки и удаления обработчиков сигналов.
sighandlestart	Отслеживание моментов вызова обработчика сигнала.
sighandlefinish	Отслеживание завершения обработчика сигнала
changepriority	Отслеживание изменения приоритета процесса
onreadyq	Отслеживание постановки нити ядра в очередь нитей, готовых к выполнению.
offreadyq	Отслеживание удаления нити ядра из очереди нитей, готовых к выполнению.

Таблица 66. События теста sysproc (продолжение)

Тест (sysproc_event)	Описание
dispatch	Отслеживание вызова системного диспетчера для планирования выполнения нити
oncpu	Отслеживание получения CPU нитью ядра.
offcpu	Отслеживание освобождения CPU нитью ядра.
blockthread	Отслеживание блокировки нити (в этом состоянии нить не может получить CPU).
foldcpu	Отслеживание свертывания ядра CPU.
bindprocessor	Отслеживание привязки процесса/нити к CPU
changecpu	Отслеживание временной смены CPU нитью ядра
resourceattach	Отслеживание событий подключения одного ресурса к другому
resourcedetach	Отслеживание событий отключения одного ресурса от другого
drphasestart	Отслеживание начала drphase
drphasefinish	Отслеживание завершения drphase

Метод доступа к данным в точке тестирования

ProbeVue позволяет обращаться к данным через встроенные переменные.

В зависимости от доступности встроенные переменные бывают трех типов:

1. Доступные в любой точке тестирования независимо от администратора тестов. Пример: `__curthread`.
2. Доступные в тестах определенного администратора тестов.
3. Доступные только в определенных тестах (интересующие события)

Администратор тестов sysproc разрешает доступ к данным через встроенные переменные типов (1) и (3). В следующей таблице показана доступность встроенных переменных типа (1). Встроенные переменные администратора тестов sysproc имеют тип данных long long.

Ниже приведен список встроенных переменных типа (1).

- __trcid
- __errno__kernelmode
- __arg1 - __arg7
- __curthread
- __curproc
- __mst
- __tid
- __pid
- __ppid
- __pgid
- __uid
- __euid
- __ublock
- __execname
- __pname

Встроенные переменные также классифицируются как контекстно зависимые и контекстно независимые. Содержимое контекстно зависимых встроенных переменных зависит от контекста выполнения теста.

Ядро AIX выполняется в контексте нити или прерывания. Контекстно зависимые тесты выдают правильный результат, когда тест запускается в контексте нити или процесса.

Результаты, получаемые из контекстно зависимых встроенных переменных в контексте выполнения прерывания, могут быть непредсказуемыми. Контекстно независимые встроенные переменные не зависят от контекста выполнения и могут безопасно использоваться в любой среде выполнения теста.

Таблица 67. Контекстно зависимые и контекстно независимые встроенные переменные

Контекстно зависимые встроенные переменные	Контекстно независимые встроенные переменные
__curthread	__trcid
__curproc	__errno
__tid	__kernelmode
__pid	__arg1 - __arg7
__ppid	__mst
__pgid	
__uid	
__euid	
__ublock	
__pname	
__execname	

Точки тестирования

Точки тестирования - это конкретные события, для которых активируется тест. Ниже приведен список точек тестирования.

forkfail

Тест `forkfail` запускается в случае сбоя функции fork. Этот тест определяет причины сбоя функции fork.

Синтаксис: `@@sysproc:forkfail:<pid/tid/*>`

Поддерживаемая специальная встроенная переменная

```
__forkfailinfo
{
fail_reason;
}
```

Переменная `fail_reason` имеет одно из следующих значений:

Таблица 68. Тест fail_reason. Причины сбоя

Причина	Описание	
FAILED_RLIMIT	Сбой из-за ограничений rlimit	
FAILED_ALLOCATIONS	Сбой из-за выделения внутренних ресурсов	
FAILED_LOADER	Сбой на этапе загрузчика	
FAILED_PROCDUP	Сбой в procdup	

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`, `__ublock`, `__execname`, `__pname`.

Среда выполнения

Выполняется в среде процесса.

Пример

В следующем примере показан мониторинг всех сбоев функции fork из-за rlimit в системе.

```
@@BEGIN
{
         x = 0;
}

@@sysproc:forkfail:*
        when (__forkfailinfo->fail_reason == FAILED_RLIMIT)
{
                printf ("процессу %s с pid %llu не удалось создать дочерний процесс функцией fork\n",__pname,__pid);
                x++;
}

@@END
{

        printf ("Обнаружено %d сбоев в течение данного сеанса vue\n",x);
}
```

execfail

Тест `execfail` запускается в случае сбоя функции `exec`. Тест `execfail` используется для определения причины сбоя.

Синтаксис: `@@sysproc:execfail:<pid/tid/*>`

Таблица 69. Тест execfail. Причины сбоя

Причина	Описание	
FAILED_PRIVILEGES	Новому процессу не удалось получить или унаследовать права доступа	
FAILED_COPYINSTR	Новому процессу не удалось скопировать инструкцию	
FAILED_V_USERACC	Новому процессу не удалось удалить области v_useracc	
FAILED_CLEARDATA	Сбой во время очистки данных для нового процесса	
FAILED_PROCSEG	Сбой при создании частного сегмента процесса	
FAILED_CH64	Не удалось преобразовать в 64-разрядный процесс	
FAILED_MEMATT	Не удалось подключиться к набору ресурсов памяти	
FAILED_SRAD	Не удалось подключиться к srad	
FAILED_MSGBUF	Нулевая длина буфера сообщения об ошибке	
FAILED_ERRBUF	Не удалось выделить буфер сообщения об ошибке	
FAILED_ENVAR	Не удалось выделить переменные среды	
FAILED_CPYSTR	Ошибка копирования строки	
FAILED_ERRBUFCPY	Не удалось скопировать сообщения об ошибке из errmsg_buf	
FAILED_TOOLNGENV	Слишком длинный блок переменных среды для выделенной памяти	
FAILED_USRSTK	Не удалось настроить стек пользователя	
FAILED_CPYARG	Не удалось скопировать список аргументов в стек	
FAILED_INITPTRACE	Не удалось инициализировать ptrace	

Примечание: К значению ошибки добавляется 64 при обнаружении ошибки загрузчика.

Другие поддерживаемые встроенные переменные

__errno__kernelmode, __arg1 - __arg7, __curthread, __curproc, __mst, __tid, __pid, __ppid, __pgid, __uid, __euid, __ublock, __execname, __pname.

Среда выполнения

Выполняется в среде процесса.

exit

Этот тест запускается при выходе из процесса. Выход также является системным вызовом и трассируется администратором системных вызовов. Тестирование системного вызова exit через администратор тестов sysproc позволяет установить природу и причины выхода. Он также позволяет установить причины, по которым нить пользователя завершилась в пространстве ядра и не вернулась в пространство пользователя.

Синтаксис: `@@sysproc:forkfail:<pid/tid/*>`

Выход из программы возможен по следующим причинам:

- При достижении условия завершения, когда программа пространства пользователя не может дальше выполняться.
- При получении сигнала завершения.

Поддерживаемая специальная встроенная переменная

```
__exitinfo{
 signo;
 returnval;
 iscore;
}
```

Где значение signo - номер сигнала, приведшего к завершению процесса, а returnval - значение, возвращенное функцией exit. Ненулевое значение signo допустимо, только если программа остановлена по сигналу.

Переменной iscore присваивается значение, когда генерируется дамп памяти в результате выхода из процесса.

Другие поддерживаемые встроенные переменные

__errno__kernelmode, __arg1 - __arg7, __curthread, __curproc, __mst, __tid, __pid, __ppid, __pgid, __uid, __euid, __ublock, __execname, __pname.

Среда выполнения

Выполняется в среде процесса.

Пример

В следующем примере показано тестирование события exit

```
echo '@@sysproc:exit:* { printf (" %s    %llu    %llu\n", __pname, __pid,__exitinfo->returnval);}' | probevue
```

Этот код генерирует следующий вывод.

```
 ksh    5833042    0
 telnetd    7405958    1
 dumpctrl    7405960    0
 setmaps    7275006    0
 termdef    7274752    0
 hostname    7274754    0
```

```
id    8257976    0
id    8257978    0
uname    8257980    0
expr    8257982    1
```

threadcreate

Тест threadcreate запускается при успешном создании нити.

Синтаксис: @@sysproc:threadcreate:<pid/tid/*>

Примечание: Указанное значение pid или tid должно быть ИД процесса или нити, создавших данную нить.

Поддерживаемая специальная встроенная переменная

```
__threadcreateinfo
{
 tid;
 pri;
 policy;
}
```

где tid - ИД новой нити, а pri - ее приоритет. policy - стратегия планирования нити.

Таблица 70. Значения стратегии для теста threadcreate

Стратегия	Описание
SCHED_OTHER	стратегия планирования AIX по умолчанию
SCHED_FIFO	стратегия планирования FIFO (очередь)
SCHED_RR	карусельная стратегия планирования
SCHED_LOCAL	стратегия планирования локальной области нити
SCHED_GLOBAL	стратегия планирования глобальной области нити
SCHED_FIFO2	FIFO с RQHEAD после короткого сна
SCHED_FIFO3	FIFO с RQHEAD все время
SCHED_FIFO4	FIFO со слабым замещением

Другие поддерживаемые встроенные переменные

__errno__kernelmode, __arg1 - __arg7, __curthread, __curproc, __mst, __tid, __pid, __ppid, __pgid, __uid, __euid, __ublock, __execname, __pname.

Среда выполнения

Выполняется в среде процесса (пользователя или kproc).

Пример

Непрерывный вывод всех процессов в системе, создающих нити. Выводятся имя процесса, ИД процесса-создателя, ИД новой нити и системное время создания.

```
echo '@@sysproc:threadcreate:*
{ printf ("%s %llu %llu %A\n",__pname,__pid,__threadcreateinfo->tid,timestamp());}' | probevue
```

Будет выведена следующая информация.

```
nfssync_kproc 5439964 23921151 Feb/22/15 09:22:38
nfssync_kproc 5439964 24052201 Feb/22/15 09:22:38
nfssync_kproc 5439964 23920897 Feb/22/15 09:22:38
nfssync_kproc 5439964 22479285 Feb/22/15 09:22:55
nfssync_kproc 5439964 23920899 Feb/22/15 09:22:55
nfssync_kproc 5439964 22479287 Feb/22/15 09:22:55
```

threadterminate

Тест запускается для завершаемой нити.

Синтаксис: `@@sysproc:threadterminate:<pid/tid/*>`

Примечание: Указанный ИД процесса или нити должен соответствовать останавливаемому процессу или нити.

Поддерживаемые специальные встроенные переменные

Нет.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса (пользователя или kproc).

Пример

Непрерывный вывод всех процессов в системе, завершающих нити. Выводятся имя процесса, ИД процесса-создателя, ИД новой нити и системное время создания.

```
# echo '@@sysproc:threadterminate:* { printf ("%s %llu %llu %A\n",__pname,__pid,__tid,timestamp());}' | probevue

Будет выведена следующая информация.
nfssync_kproc 5439964 23855555 Feb/22/15 09:59:30
nfssync_kproc 5439964 21758249 Feb/22/15 09:59:30
nfssync_kproc 5439964 23855557 Feb/22/15 09:59:30
```

threadexcept

Этот тест запускается при возникновении программной исключительной ситуации. Программная исключительная ситуация создается, когда система обнаруживает, что программа не может дальше нормально выполняться. Одни исключительные ситуации неустранимые (недопустимая команда), другие можно устранить (изменение адресного пространства).

Синтаксис: `@@sysproc:threadexcept:<pid/tid/*>`

Поддерживаемые специальные встроенные переменные

```
__threadexceptinfo
{
 pid;
 tid;
 exception;
 excpt_address
}
```

где pid - ИД процесса, получившего исключительную ситуацию, **tid** - ИД нити ядра, получившей исключительную ситуацию, excpt_address - адрес, ставший причиной данной исключительной ситуации, а exception - одно из значений, указанных в таблице.

Таблица 71. Значения exception для теста threadexcept

Исключительная ситуация	Описание
EXCEPT_FLOAT	Исключительная ситуация операции с плавающей точкой
EXCEPT_INV_OP	Недопустимый код операции
EXCEPT_PRIV_OP	Привилегированная операция в режиме пользователя
EXCEPT_TRAP	Инструкция прерывания
EXCEPT_ALIGN	Выравнивание данных или кода
EXCEPT_INV_ADDR	Недопустимый адрес
EXCEPT_PROT	Защита
EXCEPT_IO	Синхронный ввод-вывод
EXCEPT_IO_IOCC	Исключительная ситуация ввода-вывода из IOCC
EXCEPT_IO_SGA	Исключительная ситуация ввода-вывода из SGA
EXCEPT_IO_SLA	Исключительная ситуация ввода-вывода из SLA
EXCEPT_IO_SCU	Исключительная ситуация ввода-вывода из SCU
EXCEPT_EOF	Ссылка за конец файла (mmap)
EXCEPT_FLOAT_IMPRECISE	Исключительная ситуация неточного значения с плавающей точкой
EXCEPT_ESTALE_I	Исключительная ситуация устаревшего сегмента кода
EXCEPT_ESTALE_D	Исключительная ситуация устаревшего сегмента данных
EXCEPT_PT_WATCHP	Попадание в точку наблюдения ptrace

Другие поддерживаемые встроенные переменные

__errno__kernelmode, __arg1 - __arg7, __curthread, __curproc, __mst, __tid, __pid, __ppid, __pgid, __uid, __euid.

Среда выполнения

Выполняется в среде процесса или прерывания.

Примечание: Поскольку этот датчик может запускаться в контексте прерывания, встроенные переменные, которые зависят от контекста выполнения (например, __pid, __tid), могут не содержать ИД процесса или нити. Элементы специальной встроенной переменной для этого теста гарантированно содержат правильный ИД процесса или нити.

Пример

В следующем примере показана трассировка программных исключительных ситуаций, генерируемых событием теста, которое трассируется отладчиком.

```
# cat threadexcept.e
@@sysproc:threadexcept:*
{
 printf ("PID = %llu TID= %llu ИСКЛЮЧИТЕЛЬНАЯ СИТУАЦИЯ=%llu АДРЕС= %llu\n ",
__threadexceptinfo->pid,__threadexceptinfo->tid,__threadexceptinfo-
>exception,__threadexceptinfo->excpt_address);
}

Запустите сеанс отладки программы, скомпилированной с поддержкой отладки

# dbx a.out
Введите 'help' для получения справки.
Файл дампа "core" старее текущей программы (проигнорировано)
чтение символьной информации...
(dbx) stop in main
```

```
[1] остановка в функции main
(dbx) r
[1] выполнение остановлено в функции main на строке 5
    5           int a=5;

Будет выведена следующая информация.

PID = 6816134 TID= 24052015 ИСКЛЮЧИТЕЛЬНАЯ СИТУАЦИЯ=131 АДРЕС= 268436372
```

sendsig

Этот тест запускается при отправке сигнала в процесс из внешних источников (другой процесс, процесс из пространства пользователя, из потоков ядра или контекста прерывания)

Синтаксис:`@@sysproc:sendsig:<pid/*>`

```
__dispatchinfo{
 cpuid;  <- ИД CPU

 oldpid;            <- pid текущей нити
 oldtid;  <- ИД текущей нити
 oldpriority; <- приоритет текущей нити
 newpid; <- pid нового процесса, выбранного для выполнения
 newtid; <- ИД нити, выбранной для выполнения
 newpriority; <- приоритет нити, выбранной для выполнения
}
```

где pid - идентификатор целевого процесса, принимающего сигнал. Этот тест не позволяет указывать ИД нити для фильтрации результатов.

Специальные встроенные переменные

```
_sigsendinfo{
 tpid;               ◄ целевой pid
 spid;   ◄ исходный pid
 signo;        ◄ отправляемый сигнал
}
```

где `tpid` - идентификатор целевого процесса, `spid` - идентификатор источника сигнала. `spid` отличен от нуля, когда сигнал передается из пространства пользователя или контекста процесса. Идентификатор исходного процесса нулевой, если сигнал передается из контекста исключительной ситуации или прерывания. Номер сигнала содержится в `signo`.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса или прерывания.

Примечание: Поскольку этот датчик может запускаться в контексте прерывания, встроенные переменные, которые зависят от контекста выполнения (например, `__pid`, `__tid`), могут не содержать ИД процесса или нити. Элементы специальной встроенной переменной для этого теста гарантированно содержат правильный ИД процесса или нити.

Когда этот тест запускается в контексте процесса, элементы встроенной переменной, зависящие от контекста выполнения, указывают на исходный процесс. Элементы встроенной переменной, такие как `__pid`, `__tid` и `__curthread`, содержат информацию об исходном процессе. .

Пример

Непрерывный вывод источника, получателя и номера для всех сигналов.

```
echo '@@sysproc:sendsig:* {printf ("Источник=%llu Получатель=%llu Сигнал=%llu\n",__sigsendinfo->spid,__sigsendinfo->tpid,__sigsendinfo->signo);}' |
probevue
```

Будет выведена следующая информация.

```
Источник=0 Получатель=6619618 Сигнал=14
Источник=0 Получатель=8257944 Сигнал=20
Источник=0 Получатель=8257944 Сигнал=20
```

sigqueue

Этот тест запускается, когда находящийся в очереди сигнал передается в процесс.

Синтаксис:`@@sysproc:sigqueue:<pid/*>`

Специальные встроенные переменные

```
_sigsendinfo{
 tpid;                ◄ целевой pid
 spid;   ◄ исходный pid  preprocess.cp
 signo;         ◄ отправляемый сигнал
}
```

Поскольку сигналы POSIX ставятся в очередь сигналов процесса, в этом тесте нельзя указывать идентификатор нити.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Этот тест запускается в контексте процесса-отправителя. Поэтому контекстно зависимые встроенные переменные ссылаются на процесс-отправитель в этом событии теста.

Среда выполнения

Этот тест выполняется в контексте процесса.

Пример

```
echo '@@sysproc:sigqueue:*{printf ("%llu   %llu   %llu\n",__sigsendinfo->spid,__sigsendinfo->tpid,__sigsendinfo->signo);}' | probevue
```

Будет выведена следующая информация.

```
8258004   6095294   31
```

```
sigdispose
```

```
Синтаксис: @@sysproc:sigdispose:<pid/tid/*>
```

Тест запускается, когда сигнал передается в целевой процесс на обработку. Укажите ИД процесса, получившего данный сигнал, в спецификации sysprobe для фильтрации теста.

Специальные встроенные переменные

```
__sigdisposeinfo{
 tpid;                ◄ целевой pid
 ttid;           ◄ целевой tid
 signo;      ◄ сигнал, чье действие выполняется.
 fatal;        ◄ будет установлено, если процесс будет убит во время выполнения действия сигнала
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Этот тест может запускаться из контекста процесса или прерывания. При запуске из контекста прерывания данный тест может не предоставлять требуемых значений для контекстно зависимых встроенных переменных.

Пример

Непрерывный вывод идентификатора процесса, идентификатора нити, номера сигнала и информации о том, приведет ли обработка сигнала к завершению процесса, для всех процессов в системе.

```
cat sigdispose.e

@@sysproc:sigdispose:*
{
 printf ("%llu  %llu %llu %llu\n",__sigdisposeinfo->tpid,__sigdisposeinfo->ttid, __sigdisposeinfo->signo,__sigdisposeinfo->fatal);
}

Будет выведена следующая информация.

5964064  20840935 14 0
1  65539 14 0
4719084  19530213 14 0
```

sigaction

Синтаксис:`@@sysproc:sigaction:<pid/tid/*>`

Этот тест запускается при установке или замене обработчика сигнала.

Специальные встроенные переменные

```
__sigactioninfo{
 old_sighandle;            ◄ адрес функции прежнего обработчика сигнала
 new_sighandle;            ◄ адрес функции нового обработчика сигнала
 signo;             ◄ номер сигнала
 rpid;             ◄ pid запросившего процесса
}
```

`old_sighandle` будет 0, если обработчик сигнала устанавливается первый раз.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Этот тест запускается в среде процесса.

Примечание: Ядро AIX гарантирует, что одновременно только один сигнал доставляется в процесс или нить. Другой сигнал для этого процесса или нити, может быть отправлен только после завершения доставки текущего сигнала.

Пример

Отслеживание начала и завершения всех сигналов в системе:

```
@@sysproc:sighandlestart:*
{

  signal[__tid] = __sighandlestartinfo->signo;
  printf ("Обработчик сигнала по адресу 0x%x вызван для ИД нити %llu для обработки сигнала %llu\n",__sighandlestartinfo-
>sighandle,__curthread->tid,__sighandlestartinfo->signo);
}
```

```
@@sysproc:sighandlefinish:*
{

  printf ("Обработчик сигнала завершился для ИД нити %llu для сигнала %llu\n",__curthread->tid,signal[__tid]);
  delete (signal,__tid);
}

Будет выведена следующая информация.

Обработчик сигнала по адресу 0x20001d58 вызван для ИД нити 19923365 для обработки сигнала 20
Обработчик сигнала завершен для ИД нити 19923365 для сигнала 20
Обработчик сигнала по адресу 0x10003400 вызван для ИД нити 20840935 для обработки сигнала 14
Обработчик сигнала завершен для ИД нити 20840935 для сигнала 14
Обработчик сигнала по адресу 0x10002930 вызван для ИД нити 19530213 для обработки сигнала 14
Обработчик сигнала завершен для ИД нити 19530213 для сигнала 14
Обработчик сигнала по адресу 0x300275d8 вызван для ИД нити 22348227 для обработки сигнала 14
Обработчик сигнала завершен для ИД нити 22348227 для сигнала 14
Обработчик сигнала по адресу 0x20001a3c вызван для ИД нити 65539 для обработки сигнала 14
Обработчик сигнала завершен для ИД нити 65539 для сигнала 14
```

sighandlefinish

Этот тест запускается при завершении обработчика сигнала.

Синтаксис: `@@sysproc:sighandlestart:<pid/tid/*>`

Поддерживаемые специальные встроенные переменные: нет.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса. Защищенное переключение контекста запрещено на выполняющемся CPU.

changepriority

Этот тест запускается при изменении приоритета процесса. Это событие не вызывается планировщиком или диспетчером.

Синтаксис: `@@sysproc:changepriority:<pid/tid/*>`

Примечание: Изменение приоритета тоже может быть неуспешным (успех изменения приоритета не гарантирован).

Поддерживаемые специальные встроенные переменные

```
__chpriorityinfo{
 pid;
 old_priority;   <- текущий приоритет
 new_priority; <-  новый приоритет планирования нити.
}
```

Среда выполнения

Этот тест выполняется в среде процесса.

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`, `__ublock`, `__execname`, `__pname`.

Пример

Отслеживание всех процессов, чей приоритет меняется:

```
echo '@@sysproc:changepriority:* { printf ("изменение приоритета %s с %llu на %llu\n",__pname,__chpriorityinfo->old_priority,__chpriorityinfo-
>new_priority);}' | probevue

Будет выведена следующая информация.

изменение приоритета xmgc с 60 на 17
изменение приоритета xmgc с 17 на 60
изменение приоритета xmgc с 60 на 17
изменение приоритета xmgc с 17 на 60
изменение приоритета xmgc с 60 на 17
```

offreadyq

Этот тест запускается, когда нить удаляется из системной очереди выполнения.

Синтаксис: `@@sysproc:offreadyq:<pid/tid/*>`

Поддерживаемые специальные встроенные переменные

```
__readyprocinfo{
 pid;  <- ИД процесса нити, которая становится готовой к выполнению
 tid;  <- ИД нити.
 priority; <- приоритет нити
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса или прерывания.

Вариант использования: трассировка времени, которое требуется нити на выполнение операции ввода-вывода и возврат в очередь готовых к выполнению.

```
@@BEGIN
{
 printf ("           Pid      Tid      Время           Разница\n");
}

@@sysproc:offreadyq :*
{
 ready[__tid] = timestamp();
 printf ("offreadyq: %llu %llu %W\n",__readyprocinfo->pid,__readyprocinfo->tid,ready[__tid]);
}

@@sysproc:onreadyq :*
{

 if (diff_time(ready[__tid],0,MICROSECONDS))
 {
  auto:diff = diff_time (ready[__tid],timestamp(),MICROSECONDS);
  printf ("onreadyq : %llu %llu %W      %llu\n",__readyprocinfo->pid,__readyprocinfo->tid,ready[__tid],diff);
  delete (ready,__tid);
 }
}

Будет выведена следующая информация.

           Pid      Tid      Время           Разница
offreadyq: 7799280 20709717 5s 679697μs
onreadyq : 7799280 20709717 5s 679697μs      6
offreadyq: 7799280 20709717 5s 908716μs
onreadyq : 7799280 20709717 5s 908716μs      3
offreadyq: 7799280 20709717 6s 680186μs
onreadyq : 7799280 20709717 6s 680186μs      5
offreadyq: 7799280 20709717 6s 710720μs
onreadyq : 7799280 20709717 6s 710720μs      4
offreadyq: 7799280 20709717 6s 800720μs
```

```
onreadyq : 7799280 20709717 6s 800720µs      2
offreadyq: 7799280 20709717 6s 882231µs
onreadyq : 7799280 20709717 6s 882231µs      2
offreadyq: 7799280 20709717 6s 962313µs
onreadyq : 7799280 20709717 6s 962313µs      2
offreadyq: 7799280 20709717 6s 980311µs
onreadyq : 7799280 20709717 6s 980311µs      2
```

onreadyq

Этот тест запускается, когда нить ставится в системную очередь нитей, готовых к выполнению, или меняется ее положение в этой очереди.

Синтаксис: `@@sysproc:offreadyq:<pid/tid/*>`

Поддерживаемые специальные встроенные переменные

```
__readyprocinfo{
 pid;  <- ИД процесса нити, которая становится готовой к выполнению
 tid;  <- ИД нити.
 priority; <- приоритет нити
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса или прерывания.

dispatch

Этот тест запускается, когда вызывается системный диспетчер для выбора нити для выполнения на определенном CPU.

Синтаксис:`@@sysproc:dispatch:<pid/tid/*>`

Поддерживаемая специальная встроенная переменная

```
__dispatchinfo{
 cpuid;  <- CPU, на котором будет выполняться выбранная нить.
 oldpid;            <- pid текущей нити
 oldtid;  <- ИД текущей нити
 oldpriority; <- приоритет текущей нити
 newpid; <- pid нового процесса, выбранного для выполнения
 newtid; <- ИД нити, выбранной для выполнения
 newpriority; <- приоритет нити, выбранной для выполнения
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется только в среде прерывания.

Пример

Вывод ИД нити процесса прежней и выбранной нитей на CPU '0' со временем диспетчеризации относительно момента запуска сценария

```
echo '@@sysproc:dispatch:* when (__cpuid == 0){printf ("%llu %llu %W\n",__dispatchinfo->oldtid,__dispatchinfo->newtid,timestamp());}' |
probevue
```

Будет выведена следующая информация.

```
24641983 20709717 0s 48126µs
20709717 23593357 0s 48164µs
23593357 20709717 0s 48185µs
20709717 23593357 0s 48214µs
23593357 20709717 0s 48230µs
20709717 23593357 0s 48288µs
23593357 261 0s 48303µs
261 20709717 0s 48399µs
```

Пример II

Время CPU '0', израсходованное нитями между событиями диспетчеризации.

```
@@BEGIN
{
 printf ("Расход времени CPU нитями\n");
}

@@sysproc:dispatch:* when (__cpuid == $1)
{
 if (savetime[__cpuid] != 0)
  auto:diff = diff_time (savetime[__cpuid],timestamp(),MICROSECONDS);
 else
  diff = 0;
 savetime[__cpuid] = timestamp();
 printf ("%llu %llu %llu\n",__dispatchinfo->oldtid,__dispatchinfo->cpuid,diff);
}

# probevue cputime.e 6
Расход времени CPU нитями
3146085 6 0
3146085 6 9995
3146085 6 10002
3146085 6 10008
3146085 6 99988
3146085 6 100006
3146085 6 99995
3146085 6 99989
3146085 6 100010
3146085 6 100001
3146085 6 100000
3146085 6 99998
```

Как можно видеть, нить 3146085 повторно диспетчеризуется для выполнения на данном CPU с интервалом 1 с, когда нет других нитей, конкурирующих за этот CPU.

oncpu

Этот тест запускается, когда новый процесс или нить получает CPU.

Синтаксис:`@@sysproc:oncpu:<pid/tid/*>`

Где `pid` - идентификатор процесса, а `tid` - идентификатор нити процесса или нити, получающих CPU.

Поддерживаемые специальные встроенные переменные

```
__dispatchinfo{
 cpuid;  <- CPU, на котором будет выполняться выбранная нить.
 newpid; <- pid нового процесса, выбранного для выполнения
 newtid; <- ИД нити, выбранной для выполнения
 newpriority; <- приоритет нити, выбранной для выполнения
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется только в среде прерывания.

Пример

```
Вывод времени выполнения нитей процесса sysncd на всех CPU
#!/usr/bin/probevue

@@BEGIN

{

 printf ("ИД ПРОЦЕССА  ИД НИТИ  ВРЕМЯ CPU\n");

}



@@sysproc:oncpu:$1

{

 savetime[__cpuid] = timestamp();

}



@@sysproc:offcpu:$1

{

 if (savetime[__cpuid] != 0)

  auto:diff = diff_time (savetime[__cpuid],timestamp(),MICROSECONDS);

 else

  diff = 0;

 printf ("%llu %llu %llu %llu\n",

  __dispatchinfo->oldpid,

  __dispatchinfo->oldtid,

  __dispatchinfo->cpuid,

  diff);

}

# cputime.e `ps aux|grep syncd| grep -v grep| cut -f 6 -d " "`

Будет выведена следующая информация.

3735998 18612541 0 2

3735998 15663427 0 1

3735998 15073557 0 1

3735998 18743617 0 1

3735998 18874693 0 1

3735998 18809155 0 15

3735998 18940231 0 20

3735998 18547003 0 1
```

```
3735998 19267921 0 1

3735998 19071307 0 17

3735998 18678079 0 1

3735998 18481465 0 1

3735998 19202383 0 15

3735998 19005769 0 1

3735998 19136845 0 19

3735998 6160689 0 190
```

offcpu

Этот тест запускается, когда диспетчер забирает CPU у процесса или нити.

Синтаксис:`@@sysproc:dispatch:<pid/tid/*>`

Поддерживаемые специальные встроенные переменные

```
__dispatchinfo{
 cpuid;  <- CPU, на котором будет выполняться выбранная нить.
 newpid; <- pid нового процесса, выбранного для выполнения
 newtid; <- ИД нити, выбранной для выполнения
 newpriority; <- приоритет нити, выбранной для выполнения
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется только в среде прерывания.

blockthread

Этот тест запускается, когда нить блокируется (запрещается ее выполнение на всех CPU). Блокировка - это форма сна нити, в которой нить не расходует ресурсы.

Синтаксис: `@@sysproc:blockthread:*`

Поддерживаемые специальные встроенные переменные

```
__sleepinfo{
 pid;
 tid;
 waitchan;   <-- канал ожидания данного состояния сна.
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется только в среде прерывания.

foldcpu

Этот тест запускается при свертывании ядра CPU. Этот тест не запускается в контексте процесса и не должен фильтроваться по pid и tid.

Синтаксис: `@@sysproc:foldcpu:*`

Поддерживаемые специальные встроенные переменные

```
__foldcpuinfo{
 cpuid;  <- ИД логического CPU, который активировал свертывание ядра
 gpcores;         <- количество доступных ядер общего назначения (несвернутые, неисключительные).
}
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`.

Пример:

Отслеживание всех событий свертывания CPU в системе:

```
__foldcpuinfo{
 cpuid;  <- ИД логического CPU, который активировал свертывание ядра
 gpcores;         <- количество доступных ядер общего назначения (несвернутые, неисключительные).
}
```

bindprocessor

Синтаксис: `@@sysproc:bindprocessor:<pid/tid/*>`

Этот тест запускается при привязке нити или процесса к CPU. Привязка процессора - это постоянное событие, его не следует путать с временной сменой CPU.

Поддерживаемые специальные встроенные переменные

```
__bindprocessorinfo{
 ispid       <- 1, если привязка CPU к процессу; 0, если к нити
 id;     <- ИД нити или процесса.

 cpuid;

};
```

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__curthread`, `__curproc`, `__mst`, `__tid`, `__pid`, `__ppid`, `__pgid`, `__uid`, `__euid`.

Среда выполнения

Выполняется в среде процесса.

changecpu

Этот тест запускается, когда нить временно меняет CPU. Такое событие вероятно в случае однопроцессорной обработки или намеренного перемещения некоторых событий kproc на другой CPU для выполнения связанных с CPU задач (процесс xmgc перемещается между всеми CPU для управления кучами ядра).

Синтаксис:@@sysproc:changecpu:*>

Поддерживаемые специальные встроенные переменные

```
__changecpuinfo
{
 oldcpuid; <- исходный CPU
 newcpuid;  <- целевой CPU
 pid;
 tid;  <- ИД нити
}
```

Другие поддерживаемые встроенные переменные

__errno__kernelmode, __arg1 - __arg7, __curthread, __curproc, __mst, __tid, __pid, __ppid, __pgid, __uid, __euid.

Среда выполнения

Выполняется в среде процесса.

Пример

```
@@sysproc:changecpu:*

{
printf ("changecpu PID=%llu TID=%llu old_cpuid=%d new_cpuid= %d \n",
__changecpuinfo->pid,__changecpuinfo->tid,__changecpuinfo->oldcpuid,__changecpuinfo->newcpuid);

}


Будет выведена следующая информация.

changecpu PID=852254 TID=1769787 old_cpuid=26 new_cpuid= 27

changecpu PID=852254 TID=1769787 old_cpuid=-1 new_cpuid= 0

changecpu PID=852254 TID=1769787 old_cpuid=0 new_cpuid= 1

changecpu PID=852254 TID=1769787 old_cpuid=1 new_cpuid= 2
```

resourceattach

Этот тест активируется при подключении ресурса к другому ресурсу в системе.

Синтаксис:@@sysproc:resourceattach:*>

Поддерживаемые специальные встроенные переменные

```
__srcresourceinfo{
 type;
 subtype;
 id;  <- идентификатор типа ресурса
 offset;   <- смещение в случае ресурса памяти
 length;  <- длина в случае ресурса памяти
 policy;
}
__tgtresourceinfo{
```

```
 type;
 subtype;
 id;  <- идентификатор типа ресурса
 offset;   <- смещение в случае ресурса памяти
 length;  <- длина в случае ресурса памяти
 policy;
}
```

Где type и subtype могут иметь одно из следующих значений.

Таблица 72. Тест resourceattach. Значения type и subtype

Тип ресурса	Описание
R_NADA	Ничего (недопустимая спецификация)
R_PROCESS	Процесс
R_RSET	Набор ресурсов
R_SUBRANGE	Диапазон памяти
R_SHM	Общая память
R_FILDES	Файл, идентифицированный открытым файлом
R_THREAD	Нить
R_SRADID	Идентификатор SRAD
R_PROCMEM	Память процесса

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__mst`.

Среда выполнения

Выполняется в среде процесса.

resourcedetach

Этот тест активируется при отключении ресурса от другого ресурса в системе.

Синтаксис:`@@sysproc:resourcedetach:*>`

Поддерживаемые специальные встроенные переменные

```
__srcresourceinfo{
 type;
 subtype;
 id;  <- идентификатор типа ресурса
 offset;   <- смещение в случае ресурса памяти
 length;  <- длина в случае ресурса памяти
 policy;
}

__tgtresourceinfo{
 type;
 subtype;
 id;  <- идентификатор типа ресурса
 offset;   <- смещение в случае ресурса памяти
 length;  <- длина в случае ресурса памяти
 policy;
}
```

Где type и subtype могут иметь одно из следующих значений.

Таблица 73. Тест resourcedetach. Значения type и subtype

Тип ресурса	Описание
R_NADA	Ничего (недопустимая спецификация)
R_PROCESS	Процесс
R_RSET	Набор ресурсов
R_SUBRANGE	Диапазон памяти
R_SHM	Общая память
R_FILDES	Файл, идентифицированный открытым файлом
R_THREAD	Нить
R_SRADID	Идентификатор SRAD
R_PROCMEM	Память процесса

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 - __arg7`, `__mst`, `__tid`, `__pname`.

Среда выполнения

Выполняется в среде процесса.

drphasestart

Этот тест активируется при вызове обработчика динамического изменения.

Синтаксис:`@@sysproc:drphasestart:*`

Поддерживаемые специальные встроенные переменные

```
__drphaseinfo{
 dr_operation;   ◄ операция динамического изменения
 dr_flags;
 dr_phase;
 handler_rc;    ◄ всегда 0 в drphasestart
}
```

dr_operation может иметь одно из следующих значений:

- Операция динамического изменения
- DR_RM_MEM_OPER
- DR_ADD_MEM_OPER
- DR_RM_CPU_OPER
- DR_ADD_CPU_OPER
- DR_CPU_SPARE_OPER
- DR_RM_CAP_OPER
- DR_ADD_CAP_OPER
- DR_RM_RESMEM_OPER
- DR_PMIG_OPER
- DR_WMIG_OPER
- DR_WMIG_CHECKPOINT_OPER
- DR_WMIG_RESTART_OPER
- DR_SOFT_RES_CHANGES_OPER
- DR_ADD_MEM_CAP_OPER

- DR_RM_MEM_CAP_OPER
- DR_CPU_AFFINITY_REFRESH_OPER
- DR_AME_FACTOR_OPER
- DR_PHIB_OPER
- DR_ACC_OPER
- DR_CHLMB_OPER
- DR_ADD_RESMEM_OPER

Флаги динамического изменения могут быть сочетанием следующих значений:
- Флаг
- DRP_FORCE
- DRP_RPDP
- DRP_DOIT_SUCCESS
- DRP_PRE_REGISTERED
- DRP_CPU DRP_MEM DRP_SPARE
- DRP_ENT_CAP
- DRP_VAR_WGT
- DRP_RESERVE
- DRP_PMIG DRP_WMIG
- DRP_WMIG_CHECKPOINT
- DRP_WMIG_RESTART
- DRP_SOFT_RES_CHANGES
- DRP_MEM_ENT_CAP
- DRP_MEM_VAR_WGT
- DRP_CPU_AFFINITY_REFRESH
- DRP_AME_FACTOR
- DRP_PHIB
- DRP_ACC_UPDATE
- DRP_CHLMB

Другие поддерживаемые встроенные переменные

`__errno__kernelmode`, `__arg1 to __arg7`, `__tid`

Среда выполнения

Выполняется в среде процесса или прерывания.

Пример

Сценарии оболочки для ProbeVue

При работе с ProbeVue удобны следующие сценарии оболочки.

Далее перечислены полезные сценарии оболочки:

sprobevue

Этот сценарий оболочки заключает все аргументы в двойные кавычки:

```
#!/usr/bin/ksh
#
# sprobevue:
#
# Простая функция-помощник для probevue
# Заключает аргументы для probevue в двойные кавычки
#
# Формат: sprobevue <флаги probevue> <сценарий> <аргументы>
#  Не поддерживает флаги -c и -A probevue
#

usage()
{
 echo "Формат: sprobevue <флаги probevue> <сценарий> <аргументы>" >&2
 echo " Не поддерживает флаги -c и -A probevue" >&2
 exit 1
}

CMD=probevue
# Создание команды для выполнения

while getopts 'c:A:I:s:o:t:X:' zargs
do
 case $zargs in
  I|s|o|t|X) CMD="$CMD -$zargs $OPTARG"       ;;
  ?) usage
 esac
завершено

shift $(($OPTIND -1))

if [ -n "$1" ]
then
 CMD="$CMD $1"
 shift
fi

for i
do
 CMD="$CMD \"$i\""
завершено

# Выполнение команды
$CMD
```

prgrep

Этот сценарий оболочки печатает ИД процесса с данным именем процесса:

```
#/usr/bin/ksh
#
# prgrep:
#
# Простая функция-помощник для probevue
# Печатает все ИД процессов с данным именем процесса
#
#  Требуются опции для печати только одного процесса
#   для печати процесса, относящегося к определенному ИД пользователя

#
# Формат: prgrep <имя-процесса>
#        prgrep -p <ИД-процесса>
#


usage()
{
```

```
 echo "Формат: prgrep <имя-процесса>" >&2
 echo "       prgrep -p <ИД-процесса>" >&2
 exit 1
}

[ -z "$1" ] && usage

if [ $1 = "-p" ]
then
 [ -z "$2" ] && usage
 pid=$2
 export pid
 ps -e | awk 'BEGIN {pid = ENVIRON["pid"]} {if ($1 == pid) print $4}'
else
 pname=$1
 export pname
 ps -e | awk 'BEGIN {pname = ENVIRON["pname"]} {if ($4 == pname) print $1}'
fi
```

Сообщения об ошибках в ProbeVue

Как говорилось ранее, для выполнения команды **probevue** требуется привилегия. Если обычный пользователь пытается выполнить команду **probevue**, среда RBAC обнаруживает это и отказывает в выполнении команды немедленно.

```
$ probevue kernel.e
ksh: probevue: 0403-006 В доступе к выполнению отказано.
```

В разделе *Идентификация и права доступа* из книги Запуск ProbeVue показано, как разрешить обычным пользователям (кроме root) с правами и привилегиями запускать команду **probevue**.

Компилятор ProbeVue, встроенный в команду **probevue**, печатает подробные сообщения об ошибках на этапе компиляции при обнаружении синтаксических ошибок, семантических ошибок или ошибок несовместимости типов. Рассмотрим следующий сценарий:

```
/* Пример синтаксической ошибки:
 * syntaxbug.e
 */
@@BEGIN
{
        int i, j, k;

        i = 4;
        j = 22;

        k = i _ z;

        printf("k = %d\n", k);

        exit();
}
```

Приведенный сценарий содержит синтаксическую ошибку в строке 11, столбец 15, оператор присваивания. Вместо символа минус (-) по ошибке введен символ подчеркивания (_). При выполнении сценария компилятор ProbeVue обнаруживает эту ошибку и создает сообщение об ошибке.

```
# probevue syntaxbug.e
syntaxbug.e: между строка 11: столбец 15 и строка 11: столбец 15: ожидается ключ
вместо этого ключа
```

Компилятор ProbeVue также обращается к внутренним системным вызовам для проверки тестовых спецификаций в сценарии Vue. Общей ошибкой является передача в кортеж тестовых точек недопустимого ИД процесса или ИД процесса, из которого выполнен выход. Другой распространенной ошибкой является то, что забывают передавать ИД процесса в командной строке в качестве аргумента, когда сценарий ожидает его. Рассмотрим следующий сценарий:

```
/* simpleprobe.e
 */
@@syscall:$1:read:entry
{
        printf("В системном вызове чтения: ИД нити = %d\n", __tid);
        exit();
}
```

Приведенный сценарий требует ИД процесса в качестве аргумента для замены переменной '$1' в кортеже тестовых точек в строке 3. При попытке тестирования процесса, из которого выполнен выход или который не существует, ядро возвратит ошибку. Ошибка также будет выдана, если ИД процесса указывает процесс ядра или процесс инициализации. Кроме того, вам не удастся протестировать процесс, которым вы не владеете, пока у вас не будет необходимых привилегий для тестирования других процессов пользователя. Для печати имени процесса с данным ИД процесса можно воспользоваться командой **prgrep** с флагом **-p**.

Примечание: Эта команда даст пустой вывод, если указанный ИД процесса не существует.

```
# probevue simpleprobe.e 233
probevue: Этот процесс не существует.
ERR-19: Строка:3 Столбец:3 Недопустимая тестовая строка
# prgrep -p 232
#
# probevue simpleprobe.e 1
ERR-19: Строка:3 Столбец:3 Недопустимая тестовая строка
# prgrep -p 1
init
# probevue simpleprobe.e
ERR-19: Строка:3 Столбец:3 Недопустимая тестовая строка
```

Команда **probevue** также может обнаружить попытки доступа к переменным ядра пользователя без привилегий. Рассмотрим сценарий kernel.e из раздела примеров программ. Следующий пример сеанса показывает, что произойдет при попытке выполнения от имени пользователя без привилегий:

```
$ probevue kernel.e
ERR-56: Строка:93 Столбец:39 Нет прав доступа к переменным ядра
ERR-56: Строка:99 Столбец:23 Нет прав доступа к переменным ядра
ERR-56: Строка:100 Столбец:24 Нет прав доступа к переменным ядра
ERR-56: Строка:101 Столбец:25 Нет прав доступа к переменным ядра
ERR-56: Строка:102 Столбец:24 Нет прав доступа к переменным ядра
ERR-102: Строка:140 Столбец:13 Операция недопустима
ERR-46: Строка:140 Столбец:9 Недопустимое присваивание, несоответствие типов
```

После успешной компиляции сценария Vue команда **probevue** обращается к системному вызову для запуска нового сеанса ProbeVue передачей промежуточного кода, созданного компилятором. Системный вызов не удастся, если среда ProbeVue не сможет инициализировать новый сеанс ProbeVue. Для этого может быть несколько причин. Например, при запуске нового сеанса для пользователя могут быть вызваны ресурсы памяти, превышающие заданные администратором пределы. Сеансу могут потребоваться ресурсы памяти, большие допустимых для отдельного сеанса. В администраторе тестирования интервалов могут быть задействованы недопустимые функции. Из одного из тестируемых процессов может быть выполнен выход после выполнения проверки на этапе компиляции. Если запуск сеанса не удался, ядро отклонит системный вызов с возвратом однозначной 64-разрядной ошибки.

Среда ProbeVue может отменить успешно запущенный и активный сеанс ProbeVue, если во время запуска тестовых действий будет обнаружена серьезная или неисправимая ошибка. Возможные ошибки включают превышение лимитов для сеанса или пользователя (использование памяти для локальных переменных нитей и переменных списков может возрастать в ходе сеанса), превышение лимитов временной строки или области стека, обращение по индексам вне массива, попытки деления на нуль и т. д. Во всех случаях ядро возвратит однозначный 64-разрядный номер ошибки во время завершения сеанса.

При неудачном завершении сеанса при запуске или после успешного запуска команда **probevue** печатает сообщение об общей ошибке, включающее однозначный 64-разрядный номер ошибки в шестнадцатеричном формате и выполняет выход. На следующей диаграмме приведены значения некоторых общих 64-разрядных

ошибок, которые могут возвращаться ядром:

Ошибка ядра	Значение	Происходит при
0xEEEE00008C285034	Не хватает памяти при выделении основных буферов трассировки.	Запуск сеанса
0xEEEE00008C285035	Не хватает памяти при выделении вспомогательных буферов трассировки.	Запуск сеанса
0xEEEE00008C52002B	Не хватает памяти при выделении строк испытательных спецификаций.	Запуск сеанса
0xEEEE000096284122	Не хватает памяти при выделении локальной памяти нити.	Запуск сеанса
0xEEEE000081284049	В администраторе тестирования интервалов используются функции обращения к области пользователя.	Запуск сеанса
0xEEEE0000D3520022	Предел числа сеансов для обычных пользователей.	Запуск сеанса
0xEEEE000096284131	Функции **get_userstring** передан недопустимый адрес.	Выполнение испытательного действия
0xEEEE00008C520145	Превышение максимального предела нити для локальных переменных нити.	Выполнение испытательного действия

Функции событий RAS

Функции "событий RAS" являются привилегированным набором функций Vue, предусмотренных для целей отладки очень специализированных систем или приложений.

Функции "событий RAS" являются привилегированным набором функций Vue, предусмотренных для целей отладки очень специализированных систем или приложений. Они не предназначены для общего использования. Они предоставляют средства для трассировки и создания дампа системы. Многие из этих функций являются "сквозными" функциями, которые позволяют сценарию Vue непосредственно вызывать службы ядра, и, следовательно, с их применением связаны определенные риски. Для вызова этих функций в сценарии Vue требуются специальные привилегии: необходимы права пользователя root или права доступа **aix.ras.probevue.rase**.

Во избежание риска, связанного с этими функциями, передайте флаг **-K** в команду **probevue**. В противном случае эти функции просто полностью исчезают из языка Vue.

Создание записи трассировки:

Синтаксис функций Vue для создания записей трассировки системы (и трассировки LMT) аналогичен интерфейсам ядра, которые вызываются функциями Vue.

Некоторые ограничения следующие:

- Если трассировка системы не запускается или значение *hookid* не обнаруживается трассировкой системы, эти операции не создают записей трассировки системы (попытки записи трассировки LMT в общий буфер для трассировок TRCHKLx еще сохраняются, но LMT также может быть запрещена).
- Создать запись трассировки из блока **@@systrace** Vue. В этом случае вызовы функций трассировки только создают записи трассировки общего буфера LMT для трассировок TRCHKLx, если LMT разрешена.
- Эти созданные ProbeVue события трассировки протестировать невозможно; тестироваться могут только трассировки, созданные ядром и приложением.
- Необходимы привилегии, такие как права пользователя root или права доступа **aix.ras.probevue.rase**.

Для создания записей трассировки системы существуют следующие функции Vue. Все слова данных имеют тип **long long integers**:

TRCHKL0(hookID)
Трассировка без слов данных.

TRCHKL1(hookID, D1)
Трассировка с 1 словом данных.

TRCHKL2(hookID, D1,D2)
Трассировка с 2 словами данных.

TRCHKL3(hookID, D1,D2,D3)
Трассировка с 3 словами данных.

TRCHKL4(hookID, D1,D2,D3,D4)
Трассировка с 4 словами данных.

TRCHKL5(hookID, D1,D2,D3,D4,D5)
Трассировка с 5 словами данных.

void trcgenk(int channel, int hook_ID, unsigned long long data_word, int length, untyped buffer)
Трассировка буфера.

Эти функции трассировки всегда добавляют системное время к данным события. Параметр *hookid* этих функций имеет вид `0xhhhh0000`. Он не означает, что значение *hookid* должно быть константой, а только показывает, как формируется значение *hookid*.

Примечание: Устаревшие 12-битовые значения *hookid* будут использовать только левые три шестнадцатеричных числа, а четвертое число будет нулем.

Со службой ядра **trcgenk** параметр *buffer* является указателем длины данных трассировки (4096 байтов максимум). Параметр *buffer* может быть внешней переменной, например указателем ядра или приложения прикрепленных данных, или переменной сценария, например экземпляром строки или структуры Vue. Спецификация "untyped" является сокращением для этого.

Примечание: Служба ядра **trcgenk** выполняет трассировку только до трассировки системы, а не до буферов трассировки LMT.

Можно использовать ненулевой номер канала, но необходимо обеспечить, чтобы указанный канал был разрешен для трассировки. Для этой цели возвращаемое значение из команды **trace**, запустившей данную трассировку, может быть передано в сценарий Vue. Использование запрещенного канала может не дать выполнить трассировку.

Эти функции трассировки не возвращают никакое значение.

Остановка трассировки

Для остановки трассировки системы сразу после наблюдения необходимого события можно использовать `void trcoff()` в сценарии Vue. Эта функция немедленно запрещает трассировку нулевого канала. Для нормального завершения обработки трассировки необходимо еще остановить трассировку обычным образом с помощью команды **trcstop**, внешней по отношению к ProbeVue.

Можно немедленно остановить трассировки LMT и компонентов, чтобы продолжающаяся трассировка не свернула нужные данные. Соответствующие функции возобновления необходимы, поскольку нет эквивалентной командной строки для перезапуска этих трассировок. Эти новые функции Vue следующие:

```
void mtrcsuspend()
void ctsuspend()
void mtrcresume()
void ctresume()
```

Процедура **ctsuspend** останавливает трассировку всех компонентов. Эта процедура не может использоваться для избирательной остановки трассировки по компоненту. Она останавливает только трассировку компонентов, а не другую трассировку, которую мог запросить макрос CT_HOOKx, например запись трассировки системы и LMT.

Необходимо осторожно использовать эти функции управления трассировкой, так как нет сериализации, на которую влияет код трассировки ядра. Необходимо следить за тем, чтобы только один сценарий или команда влияла на трассировку одновременно.

Остановка системы

Остановить систему и создать полный дамп позволяет следующая процедура:

```
void abend(long long code, long long data_word, ...)
```

Эта процедура аналогична службе ядра **abend** за исключением того, что в ней принимаются только до 7 параметров данных (которые будут загружены в регистры с r3 по r10).

Нетипизированные параметры

В следующих прототипах функций некоторые параметры эквивалентных функций ядра типизируются неоднозначно. Компилятор Vue обычно выполняет проверку типов всех параметров, передаваемых функции Vue, но параметры, обозначенные как имеющие тип "untyped" исключаются из проверки типов. Например, с помощью этих служб ядра произвольная строка может быть передана как NULL непосредственно в ядро, но если функция Vue была определена как принимающая параметр типа String, NULL не может быть принят. Во избежание неудобства необходимости передачи пустой строки и обеспечения того, чтобы функции Vue принимали те же параметры, что и следующий интерфейс ядра, эти функции определяются как принимающие нетипизированные параметры. Нетипизированный параметр предоставляет возможность передачи NULL вместо реальной строки Vue, но необходимо внимательно задавать значения для параметров "untyped", поскольку компилятор будет принимать любой тип этого параметра.

Примечание: На самом деле в языке Vue нет спецификации переменной "untyped". Она используется только как сокращенное обозначение.

Информация, связанная с данной:

Служба ядра trcgenk

Макрокоманды для записи событий трассировки

Создание оперативного дампа:

Для создания оперативного дампа ядра используются службы ProbeVue. Службы ProbeVue похожи на соответствующие службы ядра.

Исключением из этой общей похожести является структура **ldmp_parms**, которая не подвергается воздействию на уровне сценария. Вместо этого встроенная функция **ldmp_setupparms** имеет личный экземпляр этой структуры, которая выделяется и возвращается инициатору непосредственно как 64-битовый cookie, который должен быть передан последующим службам создания оперативного дампа на его место. Эту личную структуру одновременно может использовать только один сеанс. Можно использовать другие службы создания оперативного дампа взамен синтаксиса этих дубликатов ядра. Из-за этого скрытого выделения (а также скрытых выделений, делаемых самими службами создания оперативного дампа ядра) необходимо вызывать службу ядра либо **ldmp_freeparms**, либо **livedump**, если служба ядра **ldmp_setupparms** была вызвана и возвращена успешно. В противном случае текущий сеанс будет продолжать владеть личной структурой, приводя к неудачам всех будущих вызовов **ldmp_setupparms**. После освобождения этой личной структуры она больше не сможет быть использована ее предыдущим владельцем без другого вызова **ldmp_setupparms**. Не используйте флаг LDT_POST со службой ядра **ldmp_setupparms**, поскольку это подразумевает неподдерживаемую будущую ссылку на скрытую структуру.

Типичное приложение создания оперативного дампа должно владеть скрытой структурой только в течение очень короткого времени, обычно в течение одного блока **probevue**. Скрытой структурой владеет сеанс, и ее фактически может использовать любой блок Vue в этом сеансе. Среда автоматически освободит личную структуру и другие ресурсы ядра с помощью службы ядра **ldmp_freeparms**, когда сеанс ProbeVue будет завершен.

Поскольку элементы структуры **ldmp_parms** не видны ProbeVue, те из них, которые требуют или допускают инициализацию инициатором, настраиваются дополнительными параметрами, передаваемыми версии ProbeVue службы ядра **ldmp_setupparms**.

```
long long ldmp_setupparms(строка признаков, требуемая строка признаков
untyped title,        строка заголовка дампа или NULL
untyped prefix,       строка префикса имени файла дампа или NULL
untyped func,         строка имени невыполненной функции или NULL
long long errcode,   код ошибки
int flags,           характеристики дампа
int prio )           приоритет дампа
```

Приведенная функция **ldmp_setupparms** Vue является интерфейсом службы ядра с тем же именем за исключением того, что структура **ldmp_parms** не видна вызывающему сценарию Vue. Возвращенное значение должно быть передано другим службам создания оперативного дампа как замена указателя на структуру **ldmp_parms**, несмотря на то, что оно имеет тип 64-битового целого числа.

Строка *признаков* является необходимым операндом String, тогда как строки *title*, *prefix* и *func* необязательны. Для этих трех параметров передайте String или NULL. Все значения String должны быть в сценарии Vue. Все параметры *flags* и *prio* могут быть нулевыми или значениями из файла заголовков ядра **sys/livedump.h**. Здесь должны использоваться целочисленные константы, хотя существует альтернатива.

Для параметра *flags* используются следующие значения:

```
LDT_ONEPASS  0x02 ограниченный однопроходный дамп
LDT_NOADDCOMPS 0x08 компоненты не могут добавляться обратными вызовами
LDT_NOLOG   0x10 ошибки не должны регистрироваться
LDT_FORCE   0x20 создать этот дамп
```

Поскольку дамп будет создаваться из запрещенной ProbeVue' внутренней среды, это должен быть сериализованный, синхронный, однопроходный дамп.

Для параметра *prio* допустимы следующие значения:

```
LDPP_INFO   1 информационный дамп
LDPP_CRITICAL 7 критический дамп (по умолчанию)
```

Если для параметра *prio* задано значение нуль, LDPP_CRITICAL создается службой ядра **ldmp_setupparms** по умолчанию. В скрытой структуре **ldmp_parms** будет храниться только ненулевое значение для его переопределения.

Возвращенным значением в случае успеха будет положительный cookie, представляющий владельца скрытой структуры **ldmp_parms**.

В случае неудачи возвращенное значение будет отрицательным:

Значение	Описание
EINVAL_EVM_ COOKIE	Показывает, что личная структура **ldmp_parms** недоступна.
EINVAL_EVM_ STRING	Показывает, что параметр со значением String недопустим.

Все описываемые далее функции Vue возвращают индикации неудачи аналогичным образом с отрицательны номером ошибки ядра:

Значение	Описание
EINVAL_EVM_ COOKIE	Показывает, что инициатор неправильно задал cookie, представляющий владельца личной структуры **ldmp_parms**.
EINVAL_EVM_ STRING	Показывает, что параметр со значением String недопустим.
EINVAL_EVM_ EXTID	Показывает, что параметр *extid* не поддерживается и должен быть нулевым.

Другие номера ошибок ядра могут возвращаться следующими службами ядра:

long long ldmp_freeparms (long long cookie)

После успешного возврата службы ядра **ldmp_setupparms** внутренняя структура **ldmp_parms** может быть выделена для выполнения сценария. Vue. Необходимо освободить этот ресурс, а также другие внутренние ресурсы ядра, выделенные службами, которые добавляют компоненты в дамп, созданием дампа вызовом службы ядра **livedump** или вызовом службы ядра **ldmp_freeparms**. При этом внутренняя структура **ldmp_parms** освобождается для будущего использования.

long long livedump (long long cookie)

После вызова службы ядра **ldmp_setupparms** и по крайней мере одной из различных служб, которые добавляют компоненты (и псевдокомпоненты) в дамп, создание дампа запрашивается службой **livedump**. Эта служба создает фактический оперативный дамп в файле **/var/adm/ras/livedump** согласно спецификациям, предоставленным через службу ядра **ldmp_setupparms** и другие вызванные службы создания оперативного дампа. Параметр *cookie* является cookie, возвращенным первоначальным обращением к службе ядра **ldmp_setupparms**. Возвращенным значением будет нуль, если дамп создан успешно, EINVAL_EVM_COOKIE, если cookie недопустим, и другим номером ошибки ядра, если при обработке **livedump** ядра произошла ошибка.

long long dmp_compspec(long long flags, DCF_xxx flags определенный в sys/dump.h untyped comp, добавляемый компонент (ras_block_t, name, alias и т. д.) long long cookie, cookie, возвращенный ldmp_setupparms long long extid, не поддерживаемым –, должен быть нулевым untyped p1, первый возможный параметр компонента ...); дополнительные параметры компонента

Вызовом этой службы можно добавить любой компонент, поддерживающий оперативный дамп, в оперативный дамп; эта служба по функции идентична службе ядра с тем же именем за исключением следующих случаев:

- Параметр *extid*, который разрешает возврат dmp_extid_t (long) в среду программирования ядра, не поддерживается и должен быть нулевым. В противном случае будет возвращен EINVAL_EVM_EXTID. Нет способа передачи указателя в память ProbeVue для получения этого значения, которое затем могло бы быть использовано со службой ядра **dmp_compext** и, поэтому, не поддерживается. Вместо этого можно вызывать службу **dmp_compspec** несколько раз.
- Служба ядра допускает любое число параметров p1, p2 и т. д., в которых после последнего фактического параметра должен следовать дополнительный параметр NULL для завершения списка параметров. Только функция Vue принимает максимум четыре параметра p1, p2 и т. д. Последняя также должна быть еще и нулевой для сообщения службе ядра, сколько этих параметров, поэтому на самом деле можно задать только до 3 значимых значений. Для обеспечения выполнения этого правила интерфейс будет автоматически присваивать значение нуль последнему параметру после 3 параметров переменной.
- Параметр *comp* может быть длинным (адрес ras_block_t ядра) или String, соответственно. Тип не проверяется.
- Значения флага **#define** ядра не являются частью ProbeVue.

long long ras_block_lookup(String path)

Эта функция выполняет поиск ras_block_t, соответствующего параметру имени пути к компоненту. В отсутствие более простого поиска адреса в переменной ядра можно порекомендовать вызов службы ядра **dmp_ct**, которой требуется такой адрес.

Эта функция возвратит либо адрес ядра запрошенного ras_block_t, либо NULL, если ras_block_t не найден.

Следующие функции являются примерами "сквозных" функций, которые позволяют сценарию Vue непосредственно вызывать соответствующие службы ядра. Некоторые списки содержат неиспользуемые элементы для совместимости с ядром, поэтому можно непосредственно использовать документацию по ядру. Для неиспользуемых параметров должно передаваться значение 0. Эти службы можно использовать так же, как дубликаты их ядра, за исключением того, что адрес структуры **ldmp_parms** заменяется cookie, возвращаемым из службы ядра **ldmp_setupparms**.

Как обычно, отрицательное возвращаемое значение показывает ошибку. Это может быть номер ошибки ядра из следующей службы ядра или из процедур интерфейса, если cookie или строка неверна. Следующие интерфейсы предоставляют максимальную гибкость управляемым ядром или расширением ядра оперативным дампам.

```
long long dmp_context (флаги long long flags,DCF_xxx из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long ctx_type, DMP_CTX_xxx флаги из dump.h untyped p2)
параметр, зависящий от ctx_type (NULL, mst addr, cpuid, tid)
```

```
long long dmp_ct( long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
untyped  rasb,    указатель ras_block_t компонента
long long size)   объем буфера CT для дампа или 0 для всех
```

```
long long dmp_eaddr( long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
String name,      имя cdt
untyped  addr,       первый адрес для дампа
long long size)     число байтов в дампе
```

```
long long dmp_errbuf(long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long erridx,   0 для протокола глобальных ошибок или ИД wpar
long long p2)       не используется
```

```
long long dmp_mtrc( long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long com_size, объем общих данных LMT для дампа
long long rare_size)объем данных только LMT для дампа
```

```
long long dmp_pid( long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long pid,   ИД процесса для дампа
long long p2)       не используется
```

```
long long dmp_systrace(long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long size,  объем для дампа
long long p2)       не используется
```

```
long long dmp_tid( long long flags, DCF_xxx флаги из dump.h
long long cookie, cookie, возвращаемый ldmp_setupparms
long long name,  не используется этой функцией
long long tid,   ИД нити для дампа
long long p2)       не используется
```

Примечание: После любой ошибки в приведенных процедурах необходимо вызвать службу ядра **ldmp_freeparms**; предполагается, что затем дамп будет отменен.

Следующий пример сценария создает очень маленький простой оперативный дамп. Символ ядра c_data экспортирует структуру из ядра, фактические форма и содержимое которой не имеют значения в этом примере.

```
_kernel struct {int i1; int i2; int i3; int i4;} dc_data;

@@BEGIN
{
 long long ldmp_parms;
 long long rc;

 rc = ldmp_setupparms( "дамп dc_data",
    "Мой пример дампа",  /* заголовок дампа */
    "pvdump",  /* префикс пути к дампу */
    NULL,   /* без имени функции */
    0x1122334455667788LL, /* код ошибки */
    0x10,   /* флаг LDT_NOLOG */
    0);   /* prio дампа по умолчанию */
 printf("ldmp_setupparms rc = %016llx\n", rc);
 if (rc < 0) {
  exit();
 }

 ldmp_parms = rc; /* cookie для других функций оперативного дампа */

 /*
  * Добавление 16 байтов данных ядра в пример дампа.
  * Отметим, что "dc_data" передает адрес структуры.
  */
 rc = dmp_eaddr(0, ldmp_parms, "данные-dc", dc_data, sizeof(dc_data));
 if (rc) {
  printf("неудачный dump_eaddr: %llx\n", rc);
  ldmp_freeparms(ldmp_parms);
  exit();
 }

 /*
  * Получение примера оперативного дампа.
  */
 rc = livedump(ldmp_parms);
 if (rc) {
  printf("неудачный livedump: %llx\n", rc);
 }

 exit();
}
```

Использование символов #define для флагов оперативного дампа

Следующий пример сценария оболочки (probe.dump) можно использовать, если желательно использовать фактические определенные символы для флагов оперативного дампа, а не вручную подставлять их из файлов заголовков. В этом примере соответствующие определения берутся из файлов **livedump.h** и **dump.h** и для подстановки значений перед передачей сценария ProbeVue используется препроцессор C. Этот сценарий должен удовлетворять следующим правилам:

- Не должен начинаться с комментария `#!/usr/bin/probevue`.
- Не должен использовать символы, начинающиеся с LDPP_, LDT_, DCF_ или DMP_, конфликтующие с определениями в файлах заголовков.

Не должен создавать файлы с именем **pvdump.***, поскольку следующий сценарий переопределит их.

```
#!/bin/ksh
#
# Сценарий-помощник для сценариев Vue, который должен получать
# значения различных флагов, используемых оперативным дампом.
#
# Сценарий Vue $1
# не должен содержать комментарий "#!/usr/bin/probevue", поскольку
# препроцессор C не любит его.

sed -n      \
 -e '/(/d'    \
 -e '/^#define LDPP_/p'   \
 -e '/^#define LDT_/p'   \
 -e '/^#define DCF_/p'   \
 -e '/^#define DMP_CTX_/p'  \
      /usr/include/sys/dump.h   \
      /usr/include/sys/livedump.h  \
    > pvdump.h

echo "#include \"pvdump.h\"" > pvdump.c
cat $1 >> pvdump.c
cc -P pvdump.c
/usr/bin/probevue -K pvdump.i
rm pvdump.[cih]
```

Информация, связанная с данной:

Служба ядра livedump

Функции Vue

В Vue есть много функций. Например, можно использовать функции Vue для добавления значений в список, создания трассировки динамического стека и вычисления суммы всех элементов списка.

Язык Vue поддерживает следующие функции:

Функция	Описание
add_range	Инициализация строкового диапазона.
append	Добавляет значение в список.
atoi	Преобразует строку в целое число.
avg	Возвращает среднее значение всех элементов в списке.
commitdiscard	Фиксирует или отбрасывает данные в предварительном буфере трассировки.
convert_ip4_addr	Преобразует адрес IPv4 (данные) в формат типа данных **ip_addr_t** ProbeVue.
convert_ip6_addr	Преобразует адрес IPv6 (данные) в формат типа данных **ip_addr_t** ProbeVue.
copy_kdata	Копирует данные из памяти ядра в переменную сценария Vue.
copy_userdata	Копирует данные из пользовательской памяти в переменную сценария Vue.
count	Возвращает счетчик числа элементов в списке.
diff_time	Возвращает разность между двумя временными метками.
eprintf	Форматирует и печатает данные на стандартном устройстве вывода сообщений об ошибках.
exit	Завершает сценарий Vue.
fd_fname	Получить имя файла по дескриптору файла.
fd_fstype	Получить тип файловой системы по дескриптору файла.
fd_ftype	Получить тип файла по дескриптору файла.
fd_inodeid	Получить ИД I-узла по дескриптору файла.
fd_mpath	Получить путь к точке монтирования файловой системы по дескриптору файла.
fd_path	Получить абсолютный путь к файлу по дескриптору файла.
fpath_inodeid	Получить ИД I-узла по пути к файлу.

Функция	Описание
get_function	Возвращает имя тестируемой функции.
get_kstring	Копирует данные из памяти ядра в переменную String.
get_location_point	Возвращает имя текущей точки расположения теста.
get_probe	Возвращает имя спецификации текущей точки тестирования.
get_stktrace	Возвращает трассировку стека текущей среды выполнения.
get_userstring	Копирует данные из памяти пользователя.
list	Создает и возвращает новый пустой список.
lquantize	Логарифмическое квантование значений именованного массива с последующей печатью пар ключ-значение в графическом формате.
max	Возвращает максимальный из всех элементов в списке.
min	Возвращает минимальный из всех элементов в списке.
args	Выводит имя тестируемой функции и ее аргументы
print	Печать пар ключ-значение в именованном массиве.
printf	Форматирует и копирует данные в буфер трассировки.
ptree	Выводит дерево процессов текущего процесса
quantize	Линейное квантование значений именованного массива с последующей печатью пар ключ-значение в графическом формате.
qrange	Находит номер ячейки в диапазоне и добавляет его в именованный массив.
round_trip_time	Получить сглаженное время оборота пакета соединения TCP для указанного дескриптора сокета.
set_aso_print_options	Задает флаги sort-type, sort-by и ***list-value***.
set_date_format	Изменяет формат даты.
set_range	Инициализировать линейный и степенной тип диапазона.
sockfd_netinfo	Получить локальные и удаленные порты и IP-адреса для дескриптора сокета.
startend_tentative	Указывает начало и конец раздела предварительной трассировки.
stktrace	Генерирует и печатает динамическую трассировку стека.
strstr	Возвращает подстроку строки.
sum	Возвращает сумму всех элементов в списке.
дата и время	Возвращает текущую метку времени.
trace	Копирует данные в буфер трассировки в шестнадцатеричном текстовом формате.

add_range

Назначение:

Выполняет инициализацию типа данных строкового диапазона и добавляет строки в ячейку.

Формат:

```
add_range(range_t range_data, String S1, String S2, ..., String Sn);
```

Description:

Эта процедура выполняет инициализацию `range_data` в качестве строкового диапазона и добавляет все переданные строки в одну ячейку. При первом вызове процедуры для типа данных range строки добавляются в первую ячейку. В противном случае строки добавляются в следующую ячейку.

Параметры:

`range_data`
Тип данных **range_t**.

S1, S2,...
Строки для добавления в параметр **range_data**.

append

Назначение

Добавляет значение в список.

Синтаксис

```
void append ( List listvar, long long val );
```

Описание

Функция **append** — единственная функция объединения списков в Vue. Она добавляет значение, указанное во втором параметре, к переменной списка, указанной в первом параметре. Каждый вызов **append** добавляет новое значение в список; размер списка увеличивается на единицу. Функция append может не только добавлять к списку отдельное значение, но и объединять списки.

Примечание: Добавляемое значение должно быть целым числом или списком. В противном случае, возникнет синтаксическая ошибка. Компилятор ProbeVue поддерживает все целочисленные типы C-89, и знаковые и беззнаковые. Преобразование типов не требуется.

Функция **append** не имеет возвращаемого значения.

Дополнительная информация о списочных типах — в разделе “Списочный тип” на стр. 254. В предыдущем разделе list приведен пример сценария, в котором используется функция **append**.

Параметры

Параметры	Описание
listvar	Переменная типа **list**.
val	Добавляемое значение или список.

atoi

Назначение

Преобразует строку в целое число.

Синтаксис

```
int atoi( String str );
```

Описание

Функция **atoi** возвращает целое число, чье значение представлено строкой в параметре *str*. Она читает строку до первого символа, не являющегося символом десятичной цифры, и преобразует считанные символы в соответствующее целое число. Символы пробелов в начале строки не учитываются, перед цифрами может стоять необязательный знак числа.

Функция **atoi** применяется для преобразования строк в числа во время работы сценария sprobevue оболочки, который обрамляет все параметры двойными кавычками. В следующем примере перехватывается процесс, ветвящийся быстрее ожидаемого.

```
/* Файл: ffork.e
 *
 * Формат команды: sprobevue ffork.e имя-процесса интервал
 *
 * Выполняет трассировку, когда указанный процесс ветвится быстрее
 * истечения указанного "интервала" времени. Укажите имя процесса и
 * интервал времени в мс.
 */

/* Игнорировать прочие параметры execve */
int execve(char *path);

@@BEGIN
{
 int done;
 int pid;

 pname = $1;  /* имя наблюдаемого процесса */

 /*
  * Так как используется sprobevue, необходимо преобразовать
  * строку в число (значение интервала помещено в двойные кавычки).
  */
 delta = atoi($2); /* минимальный интервал в мс между созданием ветвей процесса */
 printf("pname = %s, delta = %d\n", pname, delta);
}

@@syscall:*:execve:entry
 when (done == 0)
{
 __auto String exec[128];
 __thread int myproc;

 /* Найти выполняемый процесс */
 exec = get_userstring(__arg1, 128);

 /* Процесс найден. Установить локальную переменную нити и сбросить 'done',
  * чтобы предотвратить вход в данный тест с этого момента.
  */
 if (exec == pname) {
  pid = __pid;
  myproc = 1;
  done = 1;
  printf("Имя процесса = %s, pid = %d\n", __pname, pid);
 }
}

@@syscall:*:fork:entry
 when (thread:myproc == 1)
{
 /* old_ts инициализирована нулем */
 probev_timestamp_t old_ts, new_ts;
 unsigned long long interval;

 /* Получить текущую метку времени */
 new_ts = timestamp();

 /* Найти время предыдущего создания ветви */
 if (old_ts != 0) {
  interval = diff_time(old_ts, new_ts, MILLISECONDS);

  /* Если интервал меньше заданного, инициировать трассировку */
  if (interval < delta)
   printf("%s (%ld) создает ветви слишком часто (%d мс)\n",
     pname, __pid, interval);
 }
```

```
 /* Сохранить метку времени текущей ветви */
 old_ts = new_ts;
}

@@syscall:*:exit:entry
 when (__pid == pid)
{
 /* Перехватить функцию exit процесса и завершить сценарий */
 printf("Процесс '%s' завершен.\n", pname);
 exit();
}
```

Параметр

Параметры	Описание
str	Строка, преобразуемая в число.

avg

Назначение

Возвращает среднее значение всех элементов в списке.

Синтаксис

```
long long avg ( List listvar );
```

Описание

Функция **avg** возвращает среднее арифметическое элементов списка, указанного в параметре *listvar*.

Параметр

Параметры	Описание
listvar	Переменная типа **list**.

commit_tentative, discard_tentative

Назначение

Фиксирует или отбрасывает данные в предварительном буфере трассировки.

Синтаксис

```
void commit_tentative( String bufID );
void discard_tentative( String bufID );
```

Описание

Функция **commit_tentative** фиксирует данные трассировки, связанные с предварительным буфером трассировки, определенным в параметре *bufID*. Эта операция сохраняет данные и делает их доступными для получателя трассировки.

Функция **discard_tentative** отбрасывает все данные в предварительном буфере трассировки, указанном в параметре *bufID*. В результате, освобождаются буферы трассировки, занятые данными предварительной трассировки.

Когда данные предварительной трассировки сохраняются вместе с нормальными данными трассировки, которые раньше были предварительными, но затем их зафиксировали; все эти данные будут выдаваться

получателю трассировки в хронологическом порядке (в соответствии с метками времени). Таким образом, желательно фиксацию и отбрасывание предварительных данных выполнять как можно раньше, чтобы освободить буферы трассировки.

Все незафиксированные данные предварительной трассировки отбрасываются по завершении сеанса ProbeVue.

В разделе “Предварительная трассировка” на стр. 272 приведено подробное описание предварительной трассировки, а также пример сценария Vue, использующего предварительную трассировку.

Параметр

Параметры	Описание
bufID	Идентификатор буфера предварительной трассировки в строковом формате.

convert_ip4_addr

Назначение

Преобразует адрес IPV4 (данные) в формат типа данных IP-адрес ProbeVue.

Синтаксис

```
ip_addr_t convert_ip4_addr (unsigned int ipv4_data);
```

Описание

Функция **convert_ip4_addr** преобразует адрес IPv4 из структуры **in_addr**, заданной в файле **/usr/include/netinet/in.h**, в тип данных IP-адреса ProbeVue **ip_addr_t**. Эта функция возвращает преобразованное значение **ip_addr_t**.

Параметры

ipv4_data
Задает данные адреса ipv4, которые требуется преобразовать в формат **ip_addr_t**.

convert_ip6_addr

Назначение

Преобразует адрес IPv6 (данные) в формат типа данных IP-адрес ProbeVue.

Синтаксис

```
ip_addr_t convert_ip6_addr (int *ipv6_data);
```

Описание

Функция **convert_ip6_addr** преобразует адрес IPv6 из структуры **in6_addr**, заданной в файле **/usr/include/netinet/in.h**, в тип данных IP-адреса ProbeVue **ip_addr_t**. Эта функция возвращает преобразованное значение **ip_addr_t**.

Параметры

ipv6_data
Задает данные адреса ipv6, которые требуется преобразовать в формат **ip_addr_t**.

Следующий пример сценария выводит информацию о получателе, которому тестируемый процесс отправляет данные.

```
/* Объявление прототипа функции */
int sendto(int s, char * uap_buf, int len, int flags, char * uap_to, int tolen);

typedef unsigned int in_addr_t;

/* Объявления структур */

/* Объявление структуры in_addr */
struct in_addr {
        in_addr_t s_addr;
};

/* Объявление структуры sockaddr_in */
struct sockaddr_in {
        unsigned char sin_len;
        unsigned char sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        unsigned char sin_zero[8];
};
/* Объявление структуры in6_addr */
struct in6_addr {
        union {
                int s6_addr32[4];
                unsigned short s6_addr16[8];
                unsigned char s6_addr8[16];
        } s6_addr;
};
/* Объявление структуры sockaddr_in6 */
struct sockaddr_in6 {
        unsigned char sin6_len;
        unsigned char sin6_family;
        unsigned short sin6_port;
        unsigned int sin6_flowinfo;
        struct in6_addr sin6_addr;
        unsigned int           sin6_scope_id;  /* набор интерфейсов для области */
};


/* Печать информации об отправителе данных */
@@syscall:*:sendto:entry
{
        struct sockaddr_in6 in6;
        struct sockaddr_in in4;
        ip_addr_t ip;

                /* Копирование данных arg5 в переменную sockaddr_storage */
                /* с помощью функции Vue copy_userdata( ) */
        copy_userdata(__arg5, in4);

       /*
        * Определение типа адреса (IPv4 или IPv6) и вызов
        * соответствующей процедуры преобразования IPv4 или IPV6.
        */
       if (in4.sin_family == AF_INET)
       {
               /* Копирование данных ipv4 в структуру sockaddr_in с помощью процедуры copy_userdata */
                copy_userdata(__arg5, in4);

               /* Преобразование данных Ipv4 в формат ip_addr_t */
                ip = convert_ip4_addr(in4.sin_addr.s_addr);

               /* Печать целевого адреса и имени хоста с помощью спецификатора формата %H и %I */
               printf("It is sending the data to node %H(%I)\n", ip,ip);
```

```
        }
        else if(in4.sin_family == AF_INET6)
        {
              /* Копирование данных ipv6 в структуру sockaddr_in6 с помощью процедуры copy_userdata */
               copy_userdata(__arg5, in6);

              /* Преобразование данных Ipv6 в формат ip_addr_t */
               ip = convert_ip6_addr(in6.sin6_addr.s6_addr.s6_addr32);

              /* Печать целевого адреса и имени хоста с помощью спецификатора формата %H и %I */
              printf("It is sending the data to node %H(%I)\n",ip,ip);
        }
}
```

count

Назначение

Возвращает число элементов списка.

Синтаксис

long long count (List *listvar* **);**

Описание

Функция **count** возвращает число элементов списка, указанного в параметре *listvar*.

Дополнительные сведения о списочных типах приведены в разделе Типы данных в Vue. В предыдущем разделе list приведен пример сценария, в котором используется функция **count**.

Параметр

Параметры	**Описание**
listvar	Переменная типа **list**.

copy_kdata

Назначение

Копирует данные из памяти ядра в переменную сценария Vue.

Синтаксис

void copy_kdata(<тип> ***kaddr*****,<type>***svar***);**

Description

Функция **copy_kdata** считывает данные из памяти ядра в переменную сценария Vue. Переменная может иметь один из типов C-89, поддерживаемых Vue, кроме типов указателей. Длина копируемых данных равна размеру переменной. Например копируется 4 байта, если целевая переменная сценария Vue имеет тип int, 8 байтов, если переменная имеет тип long long, и 48 байтов, если она является массивом 12 целых чисел или int[12].

Перед тем как использовать данные в памяти ядра в выражениях или передавать в качестве параметров в функцию Vue, их необходимо скопировать.

Если во время выполнения этой функции возникает исключительная ситуация, например когда в функцию передается неверный адрес ядра, сеанс ProbeVue аварийно прерывается с сообщением об ошибке.

Параметр

kaddr Указывает адрес данных в пространстве ядра.

svar Указывает переменную сценария, в которую скопированы данные ядра. Тип переменной сценария может совпадать с типом данных ядра.

copy_userdata

Назначение

Копирует данные из пользовательской памяти в переменную сценария Vue.

Синтаксис

```
void copy_userdata( <тип>
*uaddr,<type>svar);
```

Description

Функция **copy_userdata** считывает данные из пользовательской памяти в переменную сценария Vue. Переменная может иметь один из типов C-89, поддерживаемых Vue. Длина копируемых данных равна размеру типа переменной. Например копируется 4 байта, если целевая переменная сценария Vue имеет тип int, 8 байтов, если переменная имеет тип long long, и 48 байтов, если она является массивом 12 целых чисел или int[12].

Перед использованием в выражениях и передачей в функции Vue данные пользовательского пространства должны быть скопированы.

Если во время выполнения этой функции возникает исключительная ситуация, например, когда в функцию передан неверный адрес, сеанс ProbeVue аварийно прерывается с сообщением об ошибке.

Параметр

uaddr Адрес данных в пользовательском пространстве.

svar Указывает переменную сценария, в которую скопированы пользовательские данные. Тип переменной сценария должен совпадать с типом пользовательских данных.

diff_time

Назначение

Возвращает разность между двумя временными метками.

Синтаксис

```
unsigned long long diff_time( probev_timestamp_t ts1, probev_timestamp_t ts2, intformat );
```

Описание

Функция **diff_time** возвращает разность между двумя метками времени, записанными функцией **timestamp**. Единица измерения разности — мс или мкс — определяется параметром *format*.

В разделах get_location_point и list есть примеры сценариев, использующих функцию **diff_time**.

Параметр

Параметры	Описание
ts1	Более ранняя временная метка.
ts2	Более поздняя временная метка.
format	Допустимые значения: **MILLISECONDS** Результат функции округляется до мс. **MICROSECONDS** Результат функции округляется до мкс. Значение параметра должно быть константой.

eprintf

Назначение

Форматирует и печатает данные на стандартном устройстве вывода сообщений об ошибках.

Синтаксис

```
void eprintf ( String format[ , data, ... ]);
```

Описание

Функция **eprintf** аналогична функции **printf**, за исключением того, что ее вывод направляется на стандартное устройство вывода сообщений об ошибках. Функция **eprintf** преобразует, форматирует и копирует значения параметра data в буфер трассировки в соответствии с параметром format. Как показано в описании синтаксиса, в качестве параметров data в функцию **eprintf** можно передавать переменный список аргументов. Vue поддерживает все поддерживаемые функцией **printf** спецификаторы формата из библиотеки C, за исключением спецификатора `%p`.

Функцию **eprintf** нельзя использовать для вывода переменных списочного типа. Однако переменные строкового типа можно выводить с помощью спецификатора преобразования *%s*. Переменная типа `probev_timestamp_t` выводится в числовой форме с помощью спецификатора *%lld* или *%16llx*. Переменная типа `probev_timestamp_t` выводится в формате даты с помощью спецификатора *%A* или *%W*.

Параметр

format
Одна строка, содержащая простые символы, которые копируются в буфер трассировки без изменений. Один или несколько спецификаторов преобразования, указывающие на то, как форматировать параметры данных. Дополнительная информация о спецификаторах формата приведена в описании функции **printf** в руководстве *Technical Reference: Base Operating System and Extensions, Volume 1*.

data Параметры, соответствующие спецификаторам в параметре формат.

Примечание: Предварительная трассировка с помощью функции **eprintf** недопустима.

exit

Назначение

Завершает сценарий Vue.

Синтаксис

```
void exit() ;
```

Описание

Функция **exit** завершает выполнения сценария Vue. Она выключает все тесты, активированные в сеансе динамической трассировки, отбрасывает все данные предварительной трассировки, выполняет действия, указанные в тесте **@@END** и передает все собранные данные получателю трассировки. После того, как получатель трассировки выводит данные теста **@@END**, сеанс трассировки завершается, процесс **probevue** прекращает свою работу.

Аналогичный результат можно получить, нажав сочетание клавиш Ctrl-C в терминале, где вызвана команда **probevue** (если она выполняется не в фоновом режиме). Другой вариант — послать сигнал **SIGINT** процессу **probevue** командой **kill** или системным вызовом **kill**.

В разделе list есть пример сценария, в котором применяется функция **exit**. В разделе atoi приведен пример сценария, показывающий, как завершить выполнение сценария одновременно с завершением тестируемого процесса.

Параметр

Функция **exit** не имеет параметров в отличие от функции **exit** библиотеки C.

fd_fname

Назначение

Возвращает имя файла для определенного дескриптора файла.

Синтаксис

```
char * fd_fname(int fd);
```

Описание

Эта функция возвращает имя файла для определенного дескриптора файла. Возвращаемое значение совпадает с `__file->fname` (см. описание внутренней переменной `__file` администратора тестов ввода-вывода) для того же файла.

Примечание: Эта функция требует, чтобы значение параметра `num_pagefaults` команды **probevctrl** было положительным. Если оно нулевое (или недостаточное), то функция возвращает пустую строку вместо имени файла.

Параметры

fd Значение дескриптора файла или сокета

fd_fstype

Назначение

Возвращает тип файловой системы для определенного дескриптора файла.

Синтаксис

```
int fd_fstype(int fd);
```

Описание

Эта функция возвращает тип файловой системы, которой принадлежит определенный дескриптор файла. Возвращаемые значения совпадают с `__file->fs_type` (см. описание встроенной переменной __file в администраторе тестов ввода-вывода).

Примечание: Эта функция требует, чтобы настраиваемый параметр `num_pagefaults` команды `probevctrl` был положительным. Если он нулевой (или недостаточный), функция возвращает -1 вместо типа файловой системы.

Параметры

fd Значение дескриптора файла

fd_ftype

Назначение

Возвращает тип файла по дескриптору файла.

Синтаксис

```
int fd_ftype(int fd);
```

Описание

Эта функция возвращает тип файла для дескриптора файла. Возвращаемые значения совпадают с `__file->fs_type` (см. описание встроенной переменной __file в администраторе тестов ввода-вывода).

Примечание: Эта функция требует, чтобы настраиваемый параметр num_pagefaults команды **probevctrl** был положительным. Если он нулевой (или недостаточный), то функция возвращает -1 вместо типа файла.

Параметры

fd Значение дескриптора файла

fd_inodeid

Назначение

Возвращает ИД I-узла по дескриптору файла.

Синтаксис

```
unsigned long long fd_inodeid(int fd);
```

Описание

Эта функция возвращает ИД I-узла для файла, связанного с определенным дескриптором файла. ИД I-узла - общесистемное уникальное значение типа `unsigned long long` (оно отличается от номера I-узла файловой системы и может меняться при перезагрузках системы). Это значение совпадает со значением, возвращаемым функцией `fpath_inodeid()` для того же файла.

Примечание: Эта функция требует, чтобы настраиваемый параметр num_pagefaults команды **probevctrl** был положительным. Если он нулевой (или недостаточный), то функция возвращает 0 вместо ИД I-узла.

Параметры

fd Значение дескриптора файла

fd_mpath

Назначение

Получить путь к точке монтирования файловой системы по дескриптору файла.

Синтаксис

```
char * fd_mpath(int fd);
```

Описание

Эта функция возвращает путь к точке монтирования файловой системы, которой принадлежит определенный дескриптор файла. Возвращаемое значение совпадает с `__file->mount_path` (см. описание встроенной переменной __file в администраторе тестов ввода-вывода) для того же файла.

Примечание: Эта функция требует, чтобы настраиваемый параметр num_pagefaults команды `probevctrl` был положительным. Если он нулевой (или недостаточный), то функция возвращает пустую строку вместо пути к точке монтирования.

Параметры

fd Значение дескриптора файла

fd_path

Назначение

Возвращает абсолютный путь к файлу для определенного дескриптора файла.

Синтаксис

```
path_t fd_path(int fd);
```

Описание

Эта функция возвращает абсолютный путь к файлу для определенного дескриптора файла. Возвращаемое значение имеет тип `path_t`. Возвращаемое значение совпадает с __file->path (см. описание встроенной переменной __file в администраторе тестов ввода-вывода) для того же файла.

Примечание: Эта функция требует, чтобы настраиваемый параметр num_pagefaults команды `probevctrl` был положительным. Если он нулевой (или недостаточный), то функция возвращает пустой путь, который функция `printf("%p")` выводит как пустую строку.

Параметры

fd Значение дескриптора файла

fpath_inodeid

Назначение

Возвращает ИД I-узла по пути к файлу.

Синтаксис

```
unsigned long long fpath_inodeid(String file_path);
```

Описание

Эта функция возвращает ИД I-узла по пути к файлу. ИД I-узла - общесистемное уникальное значение типа unsigned long long (оно отличается от номера I-узла файловой системы и может меняться при перезапусках системы). Если путь к файлу не существует, то сценарий Vue отклоняется командой **probevue**. Значение ИД I-узла всегда совпадает с переменной __file->inode_id в событиях тестов vfs для одного и того же файла (см. описание встроенной переменной __file администратора тестов ввода-вывода).

Примечание: Эту функцию можно использовать в любом месте сценария `Vue` (там, где разрешены функции `Vue`).

Параметры

file_path
Строковой литерал в двойных кавычках, представляющий существующий файл. Например, `"/usr/lib/boot/unix_64"`. Не может быть переменной.

get_function

Назначение

Возвращает имя функции, включающей текущий тест. Если функция **get_function** вызывается в местах interval, systrace, BEGIN и END, она возвращает пустую строку.

Синтаксис

```
String get_function ( );
```

Описание

Функция **get_function** возвращает имя тестируемой функции, то есть функции в текущей точке тестирования. Обычно имя тестируемой функции представляет собой поле кортежа, предшествующее полю расположения.

В предыдущем разделе get_probe приведен пример сценария, в котором используется функция **get_function**.

Функция **get_function** возвращает пустую строку, если вызывается из диспетчера интервальных тестов.

Параметр

Функция **get_function** не имеет параметров.

get_kstring

Назначение

Копирует данные из памяти ядра в переменную String.

Синтаксис

```
String get_kstring( char
*kaddr,intlen);
```

Description

Функция **get_kstring** считывает данные из памяти ядра в переменную типа String.

Перед использованием в выражениях и передачей в функции Vue строки памяти ядра должны быть скопированы.

Результат функции **get_kstring** должен всегда быть переменной типа String. Если в параметре **len** указано значение *-1*, будут скопированы из памяти ядра все данные до первого байта со значением NULL (это значение в языке C используется как признак конца строки). Если длина строки превышает размер переменной типа String, в переменную будет скопировано только число символов равное размеру переменной. Однако сперва необходимо всю строку, то есть до байта со значением NULL, скопировать во временный буфер строкового типа. Пользователи этой функции должны быть следить за тем, чтобы адрес

ядра указывал на строку, оканчивающуюся NULL. В противном случае может произойти переполнение временного буфера, что в свою очередь чревато аварийным завершением сеанса ProbeVue.

Максимальную длину считываемой из памяти ядра строки можно указать в параметре **len** с помощью неотрицательного значения. В этом случае копирование будет выполнено без учета байтов со значением NULL (они могут попасть в результат). Эта функция позволяет копировать длинные строки из памяти ядра более безопасно, так как копия ограничена значением параметра **len** и не вызывает переполнения внутреннего временного буфера строки ProbeVue.

Если во время выполнения этой функции возникает исключительная ситуация, например, когда в функцию передан неверный адрес ядра, сеанс ProbeVue аварийно прерывается с сообщением об ошибке.

Параметр

addr Указывает адрес данных в пространстве ядра.

len Объем копируемых данных ядра в байтах. Значение *-1* обозначает, что данные ядра являются строкой языка C, и копирование следует выполнять до байта '\0'. Использовать значение **-1** для параметра **len** следует очень осторожно.

get_location_point

Назначение

Возвращает имя текущей точки расположения теста.

Синтаксис

```
int get_location_point ( );
```

Описание

Функция **get_location_point** возвращает текущее расположение теста как смещение от точки входа включенной функции. Например, она вернет **FUNCTION_ENTRY** или 0, если точка тестирования находится в точке входа функции и значение **FUNCTION_EXIT** или -1, если она находится в точке выхода. В остальных случаях она возвращает текущий адрес смещения.

Пример использования функции **get_location_point**:

```
@@syscall:$1:read:entry, @@syscall:$1:read:exit
 {
  probev_timestamp_t ts1, ts2;
  int diff;

  if (get_location_point() == FUNCTION_ENTRY)
   ts1 = timestamp();
  else if (get_location_point() == FUNCTION_EXIT) {
   ts2 = timestamp();
   diff = diff_time(ts1, ts2, MICROSECONDS);
   printf("Время системного вызова read = %d\n", diff);
  }
 }
```

Вызов этой функции из диспетчера интервальных тестов не поддерживается.

Параметр

Функция **get_location_point** не имеет параметров.

get_probe

Назначение

Возвращает имя спецификации текущей точки тестирования.

Синтаксис

```
String get_probe ( );
```

Описание

Функция **get_probe** возвращает внутреннее представление спецификации текущей точки тестирования. Внутреннее представление точки тестирования не содержит префикса **@@** и идентификатора процесса.

Пример использования функции **get_probe**:

```
#cat get_probe.e
 @@uft:312678:*:run_prog:entry
 {
  printf("тест '%s' функции '%s'\n",  get_probe(), get_function());
 }

 #probevue get_probe.e
 function 'run_prog' probe 'uft:*:*:run_prog:entry'
```

Параметр

Функция **get_probe** не имеет параметров.

get_stktrace

Назначение

Возвращает трассировку текущего стека.

Синтаксис

```
stktrace_t get_stktrace(int level);
```

Описание

Функция **get_stktrace** возвращает трассировку стека текущей нити. Эта трассировка стека сохраняется в переменной типа **stktrace_t** или выводится с помощью спецификатора **%t** или **%T** встроенной функции ProbeVue **printf**. Параметр level указывает количество уровней трассировки, которые следует напечатать. Поведение функции **get_stktrace**, которая используется в функции **printf**, аналогична встроенной функции **stktrace**. Единственное отличие заключается в том, что по умолчанию выводится символ с адресом, если задан спецификатор **%t** и нить в состоянии выполнения. В противном случае выводится непреобразованный адрес. Также выводится полный стек CPU путем просмотра всех машинных состояний.

Пример использования функции **get_stktrace**:

```
t1 = get_stktrace(3)              // Записывает трассировку текущего стека и сохраняет ее
                                  // в переменной t1 типа stktrace_t.
printf("%t\n", get_stktrace(4));   // Печатает трассировку текущего стека до 4-го уровня включительно.
                                 // Печатает символ с адресами.
printf("%T\n", get_stktrace(4));   // Печатает трассировку текущего стека до 4-го уровня включительно.
                                  // Печатает непреобразованные адреса.
```

Параметр

Параметры	Описание
level	Указывает число уровней стека, которые следует сохранить в переменной типа **stktrace_t**. Значение -1 обозначает, что обратная цепочка стека должна пересекаться как можно глубже. По умолчанию - 0, при этом трассировка выполняется до второго уровня. При ненулевых целых положительных значениях сохраняется столько уровней, сколько указано в переменной. Максимальное значение - 240. **Примечание:** При выводе записей из нескольких mst граница mst обозначается строкой символов '-'. Эта строка также считается уровнем 1. Это значит, что число выводимых записей равно параметру уровня минус количество строк-разделителей (если параметр уровня не -1).

get_userstring

Назначение

Копирует данные из памяти пользователя.

Синтаксис

```
String get_userstring( char * addr, int len );
```

Описание

Функция **get_userstring** считывает данные из памяти пользователя в переменную типа **String**.

Перед использованием в выражениях и передачей в функции Vue данные памяти пользователя должны быть скопированы. Обычно результат функции **get_userstring** помещается в переменную типа **String**. Если в параметре *len* указано значение -1, будут скопированы все данные до первого байта со значением NULL (это значение в языке C используется как признак конца строки). Если длина строки превышает размер переменной типа **String**, в переменную будет скопировано только число символов равное размеру переменной. Однако сперва необходимо всю строку, то есть до байта со значением NULL, скопировать во временный буфер строкового типа. Пользователи этой функции должны быть следить за тем, чтобы адрес указывал на строку, оканчивающуюся NULL. В противном случае может произойти переполнение временного буфера, что в свою очередь чревато аварийным завершением сеанса ProbeVue.

Реальную длину считываемой строки можно указать в параметре *len*. В этом случае копирование продолжается, пока не будет прочитан байт NULL или пока не будет прочитано указанное количество байтов. Это позволяет считывать в переменную типа **String** нестроковые типы данных. Позднее их можно будет вывести функцией trace.

Примечание: Vue не считает байты NULL концом строки, поэтому настоящие строки в обычной ситуации не рекомендуется копировать таким способом.

Эта функция разрешена только в тестах пользовательского пространства (таких как тип тестов **uft**) и тестах диспетчера тестов **syscall**. Если во время копирования данных происходит страничная ошибка, копирование прерывается. В этом случае переменная будет содержать только те данные, которые были скопированы до ошибки. Если во время выполнения этой функции возникает исключительная ситуация, сеанс ProbeVue аварийно прерывается с сообщением об ошибке.

В разделе “Функция динамической трассировки ProbeVue” на стр. 198 есть пример сценария, использующего функцию **get_userstring**.

Примечание: Тип результата операции копирования можно изменить с помощью преобразования типов, однако в этом случае компилятор будет выдавать предупреждающее сообщение. Таким образом, функцию **get_userstring** можно использовать для копирования в пространство ProbeVue не только строк, но и данных других типов, например структур. Пример:

```
/* File: string2int.e
 *
 * Чтение целого числа, переданного в указателе, с помощью get_userstring()
```

```
 *
 */
int get_file_sizep(int *fd);

@@BEGIN
{

 int     i;
}

@@uft:$1:*:get_file_sizep:entry
{
 i = *(int *)(get_userstring(__arg1, 4));

 printf("fd = %d\n", i);
}
```

Примечание: Переменная, принимающая результат копирования, должна иметь тип String и достаточный размер. В противном случае сеанс ProbeVue может быть аварийно прерван. Значение параметра размера данных, передаваемого в **get_userstring**, не ограничено, однако максимальный объем копируемых данных не может превышать объем доступной сеансу ProbeVue памяти.

Параметр

Параметры	Описание
addr	Адрес данных в пользовательском пространстве.
len	Объем копируемых данных в байтах. Значение -1 обозначает, что данные являются строкой языка C, и копирование следует выполнять до байта '\0'. Использовать это значение параметра *len* следует очень осторожно.

list

Назначение

Создает и возвращает пустой список.

Синтаксис

```
List list ( );
```

Описание

Функция **list** является конструктором объектов списочного типа. Она возвращает пустой список и автоматически объявляет переменную типа список для приема результата. Явное объявление переменной списочного типа невозможно. Списочные переменные всегда создаются как переменные глобального класса.

Функцию **list** можно вызывать из любого блока. Если в функцию **list** передается имя существующего списка, из этого списка удаляются все элементы.

Списочные переменные можно использовать для сбора данных целочисленного типа. Элементы списка автоматически преобразуются в тип **long long** (64-разрядное целое).

Пример использования функции **list**. Предполагается, что сценарий Vue вызывается программой sprobevue оболочки, заключающей каждый параметр в двойные кавычки.

```
/* File: list.e
 *
 * Сбор статистики времени выполнения системного вызова read
 *
 * Формат: sprobevue list.e <-s|-d>
 *
```

```
 * -s - общие данные; -d - детальная информация
 */

int read(int fd, void *buf, int n);

@@BEGIN
{
 String s[10];
 int detail;
 times = list();  /* объявление и создание списка */

 /* Обработка параметров */
 s = $1;
 if (s == "-d")
  detail = 1;
 else if (s == "-s")
  detail = 0;
 else {
  printf("Формат: sprobevue list.e <-s|-d>\n");
  exit();
 }
}

@@syscall:*:read:entry
{
 /*
  * Сохранить время входа в локальной переменной нити, чтобы
  * гарантировать, что в точке read:exit теста можно будет получить значение
  * метки времени входа. Если использовать глобальную переменную, ее значение будет заменено
  * следующей нитью выполняющей вызов read.
  * Это может стать причиной неправильных результатов при вычислении разности с меткой времени read:exit,
  * так как будет использоваться не оригинальное значение.
  */

 __thread probev_timestamp_t t1;
 t1 = timestamp();
}

@@syscall:*:read:exit
 when (thread:t1 != 0)
{
 __auto t2;
 __auto long difft;

 /* Получить время выхода */
 t2 = timestamp();
 difft = diff_time(t1, t2, MICROSECONDS);

 /* Добавить значение времени в список */
 append(times, difft);

 /* Вывод детальной информации, если в сценарий передан параметр "-d" */
 if (detail)
  printf("%s (%ld) реальное время = %d micros\n", __pname, __pid, difft);
}

@@interval:*:clock:10000
{
 /* Выводить статистические данные каждые 10 с */
 printf("Число вызовов функции read = %d, общее время = %d, макс. время = %d, " +
        "мин. = %d, средн. = %d\n",
   count(times),
   sum(times),
   max(times),
   min(times),
   avg(times));
}
```

Параметр

Функция **list** не имеет параметров.

lquantize

Назначение

Выводит ключи и связанные значения именованного массива в графическом формате с логарифмическим распределением значений.

Формат:

```
void lquantize( aso-name ,int num-of-entries, int flags, sort_key_ind)
```

Описание:

Эта функция отображает записи именованного массива в графическом формате на основе логарифмического значения содержимого именованного массива. Для печати только элементов с конкретным набором ключей можно указать ключи вместе с именем переменной именованного массива в первом аргументе. С помощью ключевого слова ANY можно запретить отдельные индексы ключей и разрешить все оставшиеся индексы. См. пример в разделе с описанием функции print().

Первый параметр является обязательным; все остальные параметры указывать необязательно. Если необязательные параметры не указаны, то применяются параметры печати по умолчанию.

Параметры:

aso-name
Имя переменной именованного массива для печати. Кроме того, можно указать ключи для всех индексов в квадратных скобках. Ключевое слово ANY соответствует всем ключам индекса.

num-of-entries
Задает число записей для вывода. Это необязательный параметр. Значение 0 позволяет просмотреть все записи. Если значение не указано, то применяется параметр печати по умолчанию для сеанса. Любое отрицательное значение соответствует 0.

flags
Задает флаги sort-type, sort-by и *list-value*. Это необязательный параметр. Флаги sort-type, sort-by и list-value описаны в разделе ‘Именованный массив’. Если указано значение 0, то применяется параметр печати по умолчанию для сеанса.

sort_key_ind
Индекс ключа для сортировки вывода. Если указано значение -1, от для сортировки применяется первый ключ. Если тип первого ключа не допускает сортировку, то вывод не сортируется.

max

Назначение:

Возвращает максимальный из всех элементов в списке.

Формат:

```
long long max ( List listvar );
```

Описание:

Функция **max** возвращает значение максимального элемента списка, указанного в параметре *listvar*.

Дополнительная информация о типе данных списка приведена в разделе “Функция динамической трассировки ProbeVue” на стр. 198. В предыдущем разделе *listvar* приведен пример сценария, в котором используется функция **max**.

Параметр

Параметры	Описание
listvar	Переменная типа **list**.

min

Назначение

Возвращает минимальный из всех элементов в списке.

Синтаксис

```
long long min ( List listvar );
```

Описание

Функция **min** возвращает значение минимального элемента списка, указанного в параметре *listvar*.

В предыдущем разделе *listvar* приведен пример сценария, в котором используется функция **min**.

Параметр

listvar: Переменная типа **list**.

print_args

Назначение

Выводит текущую функцию и значения ее аргументов.

Синтаксис

```
void print_args();
```

Описание

Функция **print_args** выводит имя функции и аргументы функции в скобках, разделенные запятой. Тип аргументов при выводе будет взят из таблицы обратной трассировки. Эту функцию разрешено использовать в тестах uft/uftxlc++ и syscall/syscallx. Кроме того, функцию можно использовать в тестах, тестируемое расположение в которых задано регулярным выражением.

Параметр

Функция **print_args** не принимает параметры.

Примечание: Если таблица обратной трассировки функции не найдена в памяти (страница с ней выгружена) и контекст страничной ошибки недоступен, функция **print_args** ничего не выводит. С помощью команды `probevctrl` можно увеличить число обрабатываемых страничных ошибок и перезапустить сценарий.

print

Назначение

Выводит ключи и связанные значения именованного массива.

Синтаксис

void print (*aso-name* **, int** *num-of-entires* **,** int *flags*, **int** *sort_key_ind* **);**

Описание

Эта функция печатает элементы именованного массива, указанные в переменной aso-name. Для печати только элементов с конкретным набором ключей можно указать ключи с именем переменной именованного массива в первом аргументе. С помощью ключевого слова ANY можно запретить отдельные индексы ключей и разрешить все оставшиеся индексы.

Пример:

```
print(aso_var[0][ANY][ANY]);    // печать всех элементов с первым ключом 0 (другие ключи могут быть любыми)
print(aso_var[ANY][ANY][ANY]);  // печать всех элементов; аналогично print(aso_var);
```

Первый параметр является обязательным; все остальные параметры указывать необязательно. Если необязательные параметры не указаны, то применяются параметры печати по умолчанию.

Примечание: Функция печати не поддерживает предварительную трассировку.

Многоразмерные ключи именованных массивов отображаются в виде списка, разделенного символами '|'; значение отображается в той же строке. Если вывод занимает несколько строк, то ключ отображается на отдельной строке, а значение - на новой строке. В следующем примере показан сценарий с именованным трехмерным массивом, содержащим целочисленные значения:

```
aso1[0]["a"][2.5] = 100;
aso1[1]["b"][3.5] = 101;
print(aso1);
Вывод предыдущей функции print():
[key1               |    key2               |    key3]              |    value
0 | a | 2.5000000 | 100
1 | b | 3.5000000 | 101
```

В следующем примере применяется именованный массив с двумя ключами типа int и строковым stktrace_t.

```
aso2[0][get_stktrace(-1)] = "abc";
print(aso2);

Вывод приведенной выше функции print() будет выглядеть следующим образом:
[key1               |    key2]                                  |    value
0
                    |
                         0x100001b8
                         0x10003328
                         0x1000166c
                         0x10000c30
                         .read+0x288
                         sc_entry_etrc_point+0x4
                         .kread+0x0
                                                                |
                                                                     abc
```

Параметр

aso-name

Имя переменной именованного массива для печати. Кроме того, можно указать ключи для всех индексов в квадратных скобках. Ключевое слово ANY соответствует всем ключам индекса.

num-of-entires
Задает число записей для вывода. Это необязательный параметр. Значение 0 позволяет просмотреть все записи. Если значение не указано, то применяется параметр печати по умолчанию для сеанса. Любое отрицательное значение соответствует 0.

flags
Задает флаги sort-type, sort-by и *list-value*. Это необязательный параметр. Флаги sort-type, sort-by и list-value описаны в разделе ‘Именованный массив’. Если указано значение 0, то применяется параметр печати по умолчанию для сеанса.

sort_key_ind
Индекс ключа, применяемый для сортировки вывода. Если указано значение -1, от для сортировки применяется первый ключ. Если тип первого ключа не допускает сортировку, то вывод не сортируется.

printf

Назначение

Форматирует и копирует данные в буфер трассировки.

Синтаксис

```
void printf ( String формат[ , данные, ... ]);
```

Описание

Функция **printf** преобразует, форматирует и копирует значения параметра *данные* в буфер трассировки, руководствуясь параметром "формат". Как показано в синтаксисе, функция **printf** принимает переменное число параметров *данные*. Vue поддерживает все спецификаторы функции **printf** библиотеки C, за исключением *%p*.

Помимо спецификаторов функции `printf()` из библиотеки C язык Vue поддерживает два дополнительных: *%A* and *%W*.

%A - вывод `probev_timestamp_t` 't' в формате даты по умолчанию. Этот формат можно изменить функцией `set_date_format()`.

`%W` - вывод `probev_timestamp_t` 't' в секундах и микросекундах относительно начала сеанса `probevue`.

%p - вывод строки, соответствующей абсолютному пути к файлу указанного значения `path_t`.

%M - вывод MAC-адреса указанного значения mac_addr_t.

%I - вывод IP-адреса в десятичном формате с точками для адресов IPv4 и в 16-ричном формате с точками для адресов IPv6 указанного значения specified ip_addr_t

%H - вывод имени хоста в виде строки, в десятичном или 16-ричном формате с точками для значения ip_addr_t.

Примечание: Если IP-адрес удается найти в DNS, функция `printf` выводит имя хоста. В противном случае выводится IP-адрес в десятичном или 16-ричном формате с точками.

Функцию **printf** нельзя использовать для вывода переменных типа **list**. Но переменные типа **string** можно выводить с помощью спецификатора преобразования *%s*. Переменная типа `probev_timestamp_t` выводится в числовой форме с помощью спецификатора *%lld* или *%16llx*. Тип `probev_timestamp_t` выводится в формате даты с помощью спецификатора *%A* или *%W*.

Примеры использования функции **printf**:

```
@@BEGIN
{
 String s[128];
 int     i;
 float f;
 f = 2.3;

 s = "Тест: %d, вещественное число = %f\n";
 i = 44;

 printf(s, i, f);

 s = "Примечание:";
 printf("%s значение i (выравнивание по левому краю) = %-12d и правому = %12d\n",
    s, i, i);

 printf("Если строка формата занимает несколько строк, " +
   "можно использовать оператор '+' для объединения нескольких строк" +
   "в одну: 0x%08x\n", i);

 exit();
}
```

Параметр

format
Строка, содержащая текст, который копируется в буфер трассировки без изменений, и спецификаторы, определяющие формат параметров *данные*. Дополнительная информация о спецификаторах приведена в описании функции AIX **printf** в книге *Technical Reference: Base Operating System and Extensions, Volume 1*.

данные
Параметры, соответствующие спецификаторам в параметре *формат*.

ptree

Назначение

Печатает дерево процессов текущего процесса.

Синтаксис

```
void ptree ( int depth );
```

Описание

Функция **ptree** печатает дерево процессов, в которое входит текущий процесс. Функция печатает как родительские процессы, так и дочерние. Глубину вывода дочерних процессов можно указать как параметр функции. Функцию нельзя использовать при тестировании BEGIN и END, а также systrace. Кроме того, допускается использование этой функции в интервальных тестах, если указан PID.

Примечание: Функция ptree не выполняется в ядре сразу после вызова из теста, а ставится в очередь и выполняется в пространстве пользователя. Таким образом, если за время ожидания дерево процессов изменится, то вывод **ptree** не будет содержать нужную пользователю структуру процессов.

Пример вывода

Пример дерева процессов:

```
PID             CMD

1               init
                  |
                  V
```

```
3342460         srcmstr
                  |
                  V
3539052         inetd
                  |
                  V
7667750         telnetd
                  |
                  V
6881336          ksh
                  |
                  V
5112038          probevue
                  |
                  V
7930038            tree    <=======
6553782               |\--tree
4849828               |\--tree
6422756                   |\--tree
3408074                       |\--tree
5963846                       |\--tree
7864392                     |\--tree
7799006                        |\--tree
```

Параметр

Параметры	Описание
depth	Задает максимальную глубину, с которой **ptree** выводит дочерние процессы. Если передано значение -1, будут выведены все процессы.

quantize

Назначение:

Выводит ключи и связанные значения именованного массива в графическом формате с линейным разбиением значений.

Формат:

void quantize (*aso-name*, int *num-of-entries*, int *flags*, int *sort_key_ind*)

Описание:

Эта функция отображает записи именованного массива в графическом формате на основе линейного значения содержимого именованного массива. Для печати только элементов с конкретным набором ключей можно указать ключи с именем переменной именованного массива в первом аргументе. С помощью ключевого слова ANY можно запретить отдельные индексы ключей и разрешить все оставшиеся индексы.

Обязательным является только первый параметр - все остальные параметры указывать необязательно. Если необязательные параметры не указаны, то применяются параметры печати по умолчанию.

Параметры:

aso-name
Имя переменной именованного массива для печати. Кроме того, можно указать ключи для всех индексов в квадратных скобках. Ключевое слово ANY соответствует всем ключам индекса.

num-of-entries

Задает число записей для вывода. Это необязательный параметр. Значение 0 позволяет просмотреть все записи. Если значение не указано, то применяется параметр печати по умолчанию для сеанса. Любое отрицательное значение соответствует 0.

flags
Задает флаги sort-type, sort-by и *list-value*. Это необязательный параметр. Флаги sort-type, sort-by и list-value описаны в разделе ‘Именованный массив’. Если указано значение 0, то применяется параметр печати по умолчанию для сеанса.

sort_key_ind

Индекс ключа для сортировки вывода. Если указано значение -1, от для сортировки применяется первый ключ. Если тип первого ключа не допускает сортировку, то вывод не сортируется.

qrange

Эта процедура получает номер ячейки для диапазонов и добавляет тип данных range в качестве типа значений именованного массива.

Формат:

```
void qrange(aso[key], range_t range_data, int value);
void qrange(aso[key], range_t range_data, String value);
```

Описание:

Процедура `qrange` может найти номер ячейки для диапазонов типов Integral и String. В случае диапазона типа Integral третий аргумент должен быть целым числом. В случае диапазона типа String третий аргумент должен быть строкой. Процедура `qrange` находит номер ячейки с учетом переданного значения. Номер ячейки увеличивается для диапазона, сохраненного в именованном массиве в качестве значения.

Параметры:

`aso[key]`
Именованный массив с указанным ключом.

`range_data`
Тип данных **range_t**.

`value`
value может быть целым числом или строкой.

round_trip_time

Назначение

Возвращает сглаженное время оборота пакета соединения TCP для определенного дескриптора сокета.

Синтаксис

```
int round_trip_time(int sock_fd);
```

Описание

Функция `round_trip_time` получает сглаженное полное время ответа (srtt) для конкретного дескриптора сокета. Она возвращает правильное значение оборота пакета для дескриптора потокового сокета и -1 для недопустимого дескриптора или дескриптора непотокового сокета. Эта функция доступна только в администраторах тестов uft и syscall.

Примечание: Эта функция требует, чтобы значение настраиваемого параметра num_pagefaults команды `probevctrl` было положительным. Если указано нулевое значение, то функция возвращает -1 вместо времени оборота пакета.

Параметры

fd Значение дескриптора файла или сокета.

set_aso_print_options

Назначение

Позволяет изменить параметры печати именованных массивов по умолчанию.

Синтаксис

```
void set_aso_print_options( int num-of-entries, int flags);
```

Описание

Функция **set_aso_print_options()** изменяет параметры печати именованных массивов по умолчанию. Доступные параметры печати и их начальные значения перечислены в разделе ‘Именованный массив’. Эта функция допустима только в тесте BEGIN.

Параметр

num-of-entries
Вывести n пар ключ-значение. Если 0, выводятся все записи.

flags
Флаги **sort-type**, **sort-by**, **list-value** и **stack-raw** описаны в разделе "Именованный массив". Это необязательные параметры.

set_range

Назначение:

Инициализация линейного или степенного диапазона.

Формат:

```
void set_range(range_t range_data, LINEAR, int min, int max, int step);
void set_range(range_t range_data, POWER, 2);
```

Описание:

Предусмотрено два варианта функции *set_range*. Для инициализации данных диапазона в качестве линейного диапазона передается флаг LINEAR с параметрами *min*, *max* и *step*. Для инициализации степенного диапазона передается флаг POWER со значением 2. С учетом переданных аргументов диапазон инициализируется как линейный или степенной. Линейные диапазоны инициализируются с помощью параметров *min*, *max* и *step*, а степенные диапазоны - с помощью степени 2.

Параметры (для линейного диапазона):

range_data
Тип данных **range_t**.

LINEAR
Флаг, указывающий на линейное распределение **range_data**.

min
Указывает нижнюю границу **range_data**.

max
Указывает верхнюю границу **range_data**.

step
Задает размер диапазона для каждой строки **range_data**. Параметры min, max и step могут содержать значения только следующих типов: int, short, long, long, long. Другие типы недопустимы.

Параметры (для типа степенного диапазона):

range_data
Тип данных **range_t**.

POWER
Целочисленная константа, указывающая, что распределение значений степенное.

Константа, указывающая показатель степени. В настоящее время поддерживается только квадратичное распределение.

set_date_format

Назначение

Изменяет формат даты, который используется для вывода типа даты `probev_timestamp_t`.

Синтаксис

```
void set_date_format(String s);
```

Описание

Изменяет формат даты.

Эта функция поддерживает все спецификаторы преобразования для формата даты, поддерживаемые функцией `strftime()` из библиотеки C. Спецификаторы, не поддерживаемые функцией `strftime()`, недопустимы, и в этом случае используется формат по умолчанию.

Формат по умолчанию
MM:DD:YYYY hh:mm:ss TZ

MM Месяц года в десятичном виде (от 01 до 12).

DD День месяца в десятичном виде (от 01 до 31).

YYYY Год в десятичном виде (например, 1989).

hh Час в десятичном виде (от 00 до 23).

mm Минуты часа в десятичном виде (от 00 до 59).

ss Секунды минуты в десятичном виде (от 00 до 59).

TZ Имя часового пояса, если можно определить (например, CDT).

Примечание: Функция `set_date_format()` вызывается только в тесте `@@ BEGIN`. Строковая константа должна передаваться как *format*.

Параметры

S - строка формата даты.

sockfd_netinfo

Назначение

Возвращает локальные и удаленные порты и IP-адреса по дескриптору сокета.

Синтаксис

```
void sockfd_netinfo(int sock_fd,  net_info_t ninfo);
```

Описание

Функция `sockfd_netinfo` получает локальный IP-адрес, удаленный IP-адрес, локальный номер порта и удаленный номер порта для дескриптора входного сокета. Эта функция возвращает допустимые локальные и удаленные номера портов и IP-адреса для допустимого дескриптора сокета. Она возвращает 0 для недопустимых дескрипторов и дескрипторов несокетов.

Примечание: Эта функция требует, чтобы настраиваемый параметр num_pagefaults команды `probevctrl` был положительным и желательно не меньше 2. Если он нулевой, то функция возвращает 0 вместо информации о портах и адресах.

Параметры

fd Значение дескриптора файла или сокета.

ninfo Переменная сценария типа `net_info_t`, куда копируется 4-элементный кортеж с информацией о локальных и удаленных номерах портов и IP-адресах для дескриптора файла.

start_tentative, end_tentative

Назначение

Указывает начало и конец раздела предварительной трассировки.

Синтаксис

```
void start_tentative( String bufID );
void end_tentative( String bufID );
```

Описание

Эти функции показывают начало и конец раздела предварительной трассировки в блоке Vue. Данные, создаваемые функциями вывода трассировки в разделе предварительной трассировки, хранятся до вызова функции **commit_tentative** или **discard_tentative**, которая соответственно фиксирует или отбрасывает эти данные. Функция **end_tentative** необязательна. Если она не указана, блок конца Vue неявно подразумевает конец раздела предварительной трассировки.

Создаваемые данные предварительной трассировки определяются параметром *bufID*, который должен быть строковой константой или литералом. Сбор данных предварительной трассировки может выполняться одновременно под несколькими ИД. Данные, собранные под разными ИД, фиксируются и отбрасываются как отдельные блоки. ProbeVue поддерживает до 16 буферов предварительной трассировки в одном сеансе динамической трассировки, то есть сценарий Vue может использовать до 16 разных ИД трассировки. Блок Vue может содержать несколько разделов предварительной трассировки с разными ИД.

Параметр

Параметры	Описание
bufID	Идентификатор буфера предварительной трассировки в строковом формате.

stktrace

Назначение

Генерирует и печатает динамическую трассировку стека.

Синтаксис

```
void stktrace ( int flags, int levels );
```

Описание

Функция **stktrace** выводит трассировку стека в текущей точке тестирования. По умолчанию трассировка стека генерируется в компактной форме с адресами вызова цепочек для двух уровней. Можно использовать параметры *flags* и *levels* для изменения формата и содержимого трассировки стека. ProbeVue не может читать страницы, выгруженные из памяти, поэтому трассировка стека останавливается на первой страничной ошибке.

Функция **stktrace** не возвращает значений.

Параметр

Параметры	Описание
flags	Значение 0 включает режим работы по умолчанию; другие режимы настраиваются следующими флагами: **PRINT_SYMBOLS** Печатает имена символов вместо адресов. **GET_USER_TRACE** По умолчанию трассировка стека останавливается на границе вызова системы, если тест расположен в пространстве ядра. Этот флаг включает трассировку в пространстве пользователя на указанное число уровней (параметр *levels*). **GET_ALL_MSTS** По умолчанию трассировка стека записывается только для одного контекста (машинного состояния), где был запущен тест. Если указан этот флаг, выводится трассировка стека всей цепочки контекстов для данного CPU. Для установки нескольких флагов их необходимо объединить оператором битового 'ИЛИ' (оператор '|'). Значение параметра должно быть константой.
levels	Указывает число уровней стека, которые необходимо распечатать. Значение -1 обозначает, что обратная цепочка стека должна пересекаться как можно глубже. Значение по умолчанию 0 обозначает трассировку назад до двух уровней. **Примечание:** При выводе записей из нескольких mst граница mst обозначается строкой символов '-'. Эта строка также считается уровнем 1. Это значит, что число выводимых записей равно параметру уровня минус количество строк-разделителей (если параметр уровня не -1).

strstr

Назначение

Возвращает подстроку строки.

Синтаксис

```
String strstr( String s1, String s2 );
```

Описание

Функция **strstr** ищет первое вхождение подстроки *s2* в строку *s1* и возвращает новую строку, содержащую символы строки *s1*, начиная с подстроки. Ни *s1*, ни *s2* не изменяются. Если *s2* не встречается в *s1*, функция возвращает пустую строку.

Примечание: Поведение этой функции отличается от поведения функции **strstr** библиотеки C.

Параметр

Параметры	Описание
s1	Искомая подстрока.
s2	Строка, где выполняется поиск.

sum

Назначение

Возвращает сумму всех элементов в списке.

Синтаксис

long long sum (List *listvar* **);**

Описание

Функция **sum** возвращает сумму элементов списка, указанного в параметре *listvar*.

Параметр

Параметры	Описание
listvar	Переменная типа **list**.

дата и время

Назначение

Возвращает текущую метку времени.

Синтаксис

```
probev_timestamp_t timestamp( );
```

Описание

Функция **timestamp** возвращает текущую метку времени в абстрактном типе данных **probev_timestamp_t**. Несмотря на то что значение абстрактно, оно обладает следующими свойствами:

- При одновременном вызове несколькими CPU, значения функции равны или почти равны.
- Если **timestamp** вызывается дважды, и архитектура системы гарантирует, что второй вызов происходит позднее; возвращаемое значение будет не меньше возвращаемого значения первого вызова (при условии, что система не перезагружается между вызовами).

Значения, возвращаемые **timestamp** в разных системах, никак друг с другом не связаны. Хотя компилятор позволяет обращаться с возвращаемым значением как с 64-разрядным целым, такая практика не рекомендуется, поскольку может привести к проблемам с совместимостью.

Примечание: Вместо этой функции можно использовать переменную ядра **lbolt**, чье значение показывает число тактов с момента загрузки, или переменную ядра **time**, содержащую число секунд с начала эпохи (1 января 1970 года); если менее точные метки времени допустимы.

```
typedef long long time_t;
 __kernel time_t lbolt;    /* число тактов с момента последней загрузки */
 __kernel time_t time;     /* отображенное в память число секунд с начала эпохи */
```

Параметр

Функция **timestamp** не имеет параметров.

trace

Назначение

Копирует данные в буфер трассировки в шестнадцатеричном текстовом формате.

Синтаксис

void trace (*data* **);**

Описание

Функция **trace** принимает один параметр (он должен быть переменной). **trace** не принимает выражения.

Функция **trace** копирует значение переданной переменной в буфер трассировки. Аргумент может быть любого типа, размер копируемых данных определяется по размеру объекта данных переданной переменной. То есть для целых чисел копируется 4 байта, для указателей — 4 или 8 байт (в зависимости от режима процессора — 32- или 64-разрядный), для типа **struct** — число байт равное размеру структуры. Для переменных типа **String** число копируемых байт определяется объявленной длиной строки (она не всегда совпадает с фактической длиной строки в переменной). Переменные типа **probev_timestamp_t** имеют размер не менее 8 байт.

Генератор отчетов трассировки показывает шестнадцатеричные данные, записанные функцией **trace**, группами по 4 символа без дополнительного форматирования.

Примечание: Функция **trace** принимает также переменные типа **list**, но в этом случае ее выходные данные бесполезны.

Параметр

Параметры	**Описание**
данные	Данные для копирования в буфер трассировки.

Программирование с поддержкой нескольких нитей

В этом разделе приведены инструкции по написанию программ с несколькими нитями с использованием библиотеки нитей (**libpthreads.a**).

Библиотека нитей AIX основана на стандарте X/Open Portability Guide Issue 5. Содержимое этой главы посвящено реализации стандарта XPG5 в AIX.

Распараллеливание программ позволяет использовать преимущества многопроцессорных систем, сохраняя двоичную совместимость с существующими однопроцессорными системами. Средства параллельного программирования основываются на новой концепции - нитях.

У параллельного программирования есть следующие основные преимущества перед последовательными технологиями:

- Параллельное программирование позволяет повысить производительность программы.
- Для параллельного программирования подходят многие распространенные программные модели.

В большинстве случаев параллельность достигается путем созданием нескольких однонитевых процессов, но в некоторых программах можно достичь и более высокого уровня параллельности. В процессах с несколькими нитями параллельность реализуется внутри процесса, однако при этом также применяются многие принципы программирования с несколькими однонитевыми процессами.

Перечисленные ниже разделы содержат описание нитей и относящихся к ним средств программирования. Кроме того, в них обсуждаются общие принципы распараллеливания программ:

Примечание: В этом наборе разделов термин *нить* по умолчанию означает *пользовательскую нить*. Это утверждение не распространяется на разделы, посвященные программированию ядра.

Сведения о нитях и процессах

Нить - это независимый поток управления, работающий в том же адресном пространстве, что и остальные независимые потоки управления в рамках процесса.

Обычно характеристики нити и процесса объединены в объект, называемый *процессом*. В других операционных системах нити иногда называются *простыми процессами*, а в некоторых системах принято несколько отличающееся понятие *нити*.

Ниже описаны различия между нитью и процессом.

В традиционных системах, применяющих процессы с одной нитью, у процесса есть набор свойств. В системах с несколькими нитями эти свойства распределены между процессами и нитями.

Применение нитей имеет некоторые ограничения, поэтому некоторые задачи приходится решать с помощью программ с несколькими процессами.

Понятия, связанные с данным:

“Область действия и уровень параллелизма” на стр. 451
Область действия определяет способ установки соответствия между пользовательской нитью и нитью ядра.

“Порождение и завершение процессов” на стр. 468
Так как любой процесс содержит хотя бы одну нить, создание (т.е. порождение) и завершение процессов подразумевает создание и завершение нитей.

Свойства процесса

Процесс в системе с несколькими нитями - это непостоянный объект.

Его следует понимать как среду для выполнения действий. Процесс обладает всеми традиционными атрибутами:

- ИД процесса, ИД группы процессов, ИД пользователя и ИД группы
- Среда
- Рабочий каталог

Кроме того, у процесса есть общее адресное пространство и общие системные ресурсы:

- Дескрипторы файлов
- Действия по обработке сигналов
- Общие библиотеки
- Средства связи между процессами (например, очереди сообщений, конвейеры, семафоры и общие области памяти)

Свойства нити

Нить - это планируемая единица работы. Нить - это планируемая единица работы.

Она обладает только теми свойствами, которые необходимы для ее независимой работы. К этим свойствам относятся следующие:

- Стек
- Свойства планирования (например, стратегия и приоритет)
- Набор ожидающих и блокированных сигналов
- Данные, относящиеся к конкретной нити

Пример данных, относящихся к конкретной нити, - это индикатор ошибки **errno**. В системах с несколькими нитями **errno** - это не глобальная переменная, а функция, возвращающая значение **errno** для конкретной нити. В разных системах **errno** может быть реализована по-разному.

Нити в рамках одного процесса не следует рассматривать как группу процессов. Все нити обрабатываются в одном и том же адресном пространстве. Это означает, что два указателя из различных нитей с одинаковыми значениями указывают на одни и те же данные. Кроме того, если нить изменяет какой-либо из общих ресурсов системы, изменения отражаются на всех остальных нитях процесса. Например, если нить закрывает файл, он становится закрытым для всех нитей.

Главная нить

При создании процесса автоматически создается одна нить, называемая *главной нитью*.

Благодаря этому обеспечивается совместимость между старыми процессами с одной (неявной) нитью и новыми процессами с несколькими нитями. Главная нить обладает некоторыми особыми свойствами, невидимыми для программиста и предназначенными для обеспечения двоичной совместимости между старыми программами с одной нитью и новыми операционными системами, поддерживающими обработку нескольких нитей. В этой же нити выполняется **основная** процедура программы с несколькими нитями.

Модульность

Программы часто создаются как набор отдельных частей, взаимодействующих друг с другом для получения нужного результата.

Программа может представлять собой единую сложную структуру, выполняющую несколько функций с помощью различных частей программы. Однако более простым решением является создание нескольких модулей, которые совместно используют системные ресурсы, но выполняют каждый свою функцию.

В программе, состоящей из нескольких модулей, каждый модуль имеет собственное назначение и выполняет отдельный набор действий. Эти модули независимы друг от друга, они взаимодействуют только при передаче информации. При этом для обеспечения целостности данных необходимо синхронизировать работу модулей.

Нити отвечают всем требованиям, предъявляемым при модульном программировании. Нити позволяют использовать общие данные (все нити процесса совместно используют одно адресное пространство) и надежные средства синхронизации, такие как взаимные блокировки и переменные условий.

Программные модели

В этом разделе рассмотрены различные модели программного обеспечения.

Все эти модели требуют создания модульных программ. Для эффективного решения сложных задач модели можно комбинировать.

Эти модели можно применять как для реализации стандартных решений с помощью нескольких процессов, так и для решения задач с помощью одного процесса с несколькими нитями в системах с поддержкой

нескольких нитей. В приведенных ниже описаниях слово *модуль* обозначает либо *процесс* с одной нитью, либо отдельную *нить* в процессе с несколькими нитями.

С помощью нитей можно реализовать следующие общие программные модели:

Модель главный-подчиненный элемент:

В модели главный-починенный элемент главный модуль после получения одного или нескольких запросов создает подчиненные элементы для их выполнения. Обычно, главный элемент определяет количество и назначение подчиненных элементов. Каждый из подчиненных элементов выполняется независимо от остальных.

Примером данной модели может являться программа буферизации задания печати, управляющая группой принтеров. Задача программы буферизации - обеспечить последовательную обработку полученных запросов на печать. При получении запроса главный модуль выбирает принтер и передает подчиненному элементу команду печати задания на этом принтере. Каждый подчиненный модуль за один вызов печатает на одном принтере одно задание, управляя потоком данных и другими параметрами печати. Программа буферизации может поддерживать отмену обработки заданий или другие функции, которые требуют прерывания работы подчиненных модулей или переназначения заданий.

Модели разделения действий:

В модели разделения действий (иногда она называется *моделью с одновременным вычислением* или *моделью рабочей команды*) несколько модулей параллельно выполняют одну задачу. Здесь не существует главных элементов, все модули работают параллельно и независимо друг от друга.

Примером модели разделения действий может служить следующая параллельная реализация команды **grep**. Сначала команда **grep** определяет пул файлов для просмотра. После этого она создает группу модулей. Каждый модуль обрабатывает отдельный файл из этого пула и ищет в нем заданный шаблон, передавая результаты на общее устройство вывода. После просмотра выделенного ему файла модуль получает другой файл из пула или завершает работу, если пул пуст.

Модели изготовитель-потребитель:

Модель изготовитель-потребитель (иногда она называется *конвейером*) обычно применяется в производстве. Эта модель подразумевает, что некий продукт проходит несколько этапов обработки от простого объединения исходных компонентов до стадии готовности.

Обычно такой продукт изменяется и передается на следующий этап обработки одним рабочим. В компьютерных терминах удачным примером этой модели могут служить команды конвейерной обработки AIX, например **cpio**.

Например, модуль чтения получает данные непосредственно из потока стандартного ввода и передает их обработчику, который после обработки передает данные модулю записи, направляющему их в поток стандартного вывода. Параллельное программирование позволяет выполнять эти действия одновременно: пока модуль записи передает обработанные данные в поток вывода, модуль чтения может получать новую порцию данных.

Нити ядра и пользовательские нити

Нить ядра - это единица планирования. Такими нитями управляет системный планировщик.

Разбиение процесса на нити, с которыми работает системный планировщик, в существенной степени зависит от реализации. Для создания переносимых программ в стандартных библиотеках реализованы *пользовательские* нити.

Нить ядра - это элемент ядра, такой же, как процессы или обработчики прерываний. Подобными элементами управляет системный планировщик. Нить ядра обрабатывается в рамках процесса, но на нее может ссылаться любая другая нить в системе. Программист не может управлять такими нитями непосредственно, если только он не разрабатывает расширения ядра или драйверы устройств. Дополнительная информация о программировании процессов ядра приведена в разделе *Kernel Extensions and Device Support Programming Concepts*.

Пользовательская нить - это элемент, применяемый программистами для параллельного выполнения различных операций в одной программе. API для обработки пользовательских нитей содержатся в специальной *библиотеке нитей*. Пользовательская нить существует только в рамках процесса; пользовательская нить из процесса *A* не может ссылаться на пользовательскую нить из процесса *B*. Библиотека нитей работает с нитями ядра с помощью собственного интерфейса, позволяющего запускать пользовательские нити. API для пользовательских нитей, в отличие от интерфейса нитей ядра, являются частью модели создания переносимых программ, соответствующей стандартам POSIX. Поэтому программы с несколькими нитями, разработанные в одной системе под управлением AIX, можно легко перенести в другую систему.

В других операционных системах применяется другая терминология: пользовательские нити называются *простыми нитями*, а нити ядра - *простыми процессами*.

Модели обработки нитей и виртуальные процессоры

Библиотека нитей устанавливает соответствие между пользовательскими нитями и нитями ядра. Способ установки такого соответствия называется *моделью обработки нитей*.

Существует три различных модели обработки нитей, соответствующих трем различным способам установления соответствия между пользовательскими нитями и нитями ядра:

- Модель M:1
- Модель 1:1
- Модель M:N

Соответствие между пользовательскими нитями и нитями ядра устанавливают *виртуальные процессоры*. Виртуальный процессор (VP) - это элемент библиотеки, как правило, неявный. Для пользовательской нити виртуальный процессор играет ту же роль, что и CPU. В библиотеке виртуальный процессор является нитью ядра или структурой, связанной с нитью ядра.

В модели M:1 всем пользовательским нитями ставится в соответствие одна нить ядра; все пользовательские нити выполняются на одном VP. Это соответствие устанавливает планировщик библиотеки. Все средства программирования пользовательских нитей используются только библиотекой. Эта модель допустима для любых систем, в частности, для традиционных систем, не предусматривающих обработки нескольких нитей.

В модели 1:1 для каждой пользовательской нити выделяется отдельная нить ядра; каждая пользовательская нить выполняется на отдельном VP. Большая часть средств программирования пользовательских нитей применяется непосредственно нитями ядра. Эта модель применяется по умолчанию.

В модели M:N всем пользовательским нитям ставится в соответствие пул нитей ядра; все пользовательские нити выполняются пулом виртуальных процессоров. Отдельная пользовательская нить может быть связана с определенным виртуальным процессором, как в модели 1:1. Все несвязанные пользовательские нити совместно выполняются на оставшихся виртуальных процессорах. Эта модель обработки нитей самая эффективная и самая сложная; средства программирования пользовательских нитей используются как библиотекой нитей, так и нитями ядра. Для применения этой модели нужно присвоить переменной среды `AIXTHREAD_SCOPE` значение **P**.

API библиотеки нитей

В этом разделе приведена общая информация об API библиотеки нитей.

Эти сведения не требуются для написания программ с несколькими нитями, однако полезны для лучшего понимания работы API библиотеки нитей.

Объектно-ориентированный интерфейс

В API библиотеки нитей реализован объектно-ориентированный интерфейс. Программист работает с объектами с помощью указателей и других универсальных идентификаторов; детали реализации объекта скрыты от него.

Благодаря этому обеспечивается переносимость программ с несколькими нитями для систем, поддерживающих эту библиотеку нитей, а также возможность переноса данных между разными версиями AIX с перекомпиляцией только этих программ. Хотя определения некоторых типов данных содержатся в библиотечном файле заголовка (**pthread.h**), программы не должны работать с содержимым структур непосредственно, на основе этих определений, поскольку они зависят от реализации. Для работы с объектами всегда должны применяться стандартные процедуры библиотеки нитей.

В библиотеке нитей в основном используются следующие типы объектов (непрозрачных типов данных): нити, взаимные блокировки, блокировки чтения-записи и условные переменные. У этих объектов есть атрибуты, задающие свойства объектов. При создании объекта необходимо задать его атрибуты. В библиотеке нитей эти атрибуты создания сами являются объектами, называемыми *объектами атрибутов* нитей.

В библиотеке нитей предусмотрены средства для работы со следующими парами объектов:

- Объекты нитей и их атрибутов
- Объекты взаимных блокировок и их атрибутов
- Объекты условных переменных и их атрибутов
- Блокировки для чтения и записи

При создании объекта атрибутов их значения устанавливаются по умолчанию. Затем можно изменить значения отдельных атрибутов с помощью функций. Это позволяет гарантировать, что введение новых атрибутов и изменение их реализации не повлияет на программу с несколькими нитями. Следовательно, объект атрибутов можно использовать для создания одного или нескольких объектов, а затем уничтожить, причем уничтожение объекта атрибутов никак не отразится на объектах, созданных с его помощью.

Объекты атрибутов позволяют также работать с классами объектов. Для каждого класса объектов можно определить один объект атрибутов. Для создания экземпляра класса объекта следует создать такой объект с помощью объекта атрибутов класса.

Соглашение о присвоении имен в библиотеке нитей

Для идентификаторов, используемых библиотекой нитей, действует соглашение о присвоении имен. Все идентификаторы библиотеки нитей начинаются с префикса **pthread_**.

Использовать этот префикс в пользовательских именах нельзя. После префикса указывается имя компонента. В библиотеке нитей определены следующие компоненты:

Компонент	Описание
pthread_	Сами нити и различные функции
pthread_attr	Объекты атрибутов нитей
pthread_cond	Условные переменные
pthread_condattr	Объекты условных атрибутов
pthread_key	Ключи данных для конкретных нитей
pthread_mutex	Взаимные блокировки
pthread_mutexattr	Объекты атрибутов взаимных блокировок

Идентификаторы типов данных заканчиваются символом **_t**. Имена функций и макросов заканчиваются символом подчеркивания _, после которого указывается имя, обозначающее действие, которое выполняет

данная функция или макрос. Например, **pthread_attr_init** - идентификатор библиотеки нитей (**pthread_**), относящийся к объекту атрибутов нити (**attr**) и обозначающий функцию его инициализации (**_init**).

Явные имена макросов состоят из прописных букв. Однако некоторые функции могут быть реализованы как макросы, хотя их имена и состоят из строчных букв.

Файлы реализации pthread

В этом разделе рассмотрены файлы реализации **pthread**.

Реализация объектов pthread содержится в следующих файлах AIX:

Реализация	Описание
/usr/include/pthread.h	Файл заголовка C/C++, в котором содержится большинство определений объектов pthread.
/usr/include/sched.h	Файл заголовка C/C++, содержащий некоторые определения системы планирования.
/usr/include/unistd.h	Файл заголовка C/C++, содержащий определение функции **pthread_atfork()**.
/usr/include/sys/limits.h	Файл заголовка C/C++, содержащий определения некоторых объектов pthread.
/usr/include/sys/pthdebug.h	Файл заголовка C/C++, в котором содержится большинство определений отладочных объектов pthread.
/usr/include/sys/sched.h	Файл заголовка C/C++, содержащий некоторые определения системы планирования.
/usr/include/sys/signal.h	Файл заголовка C/C++, в котором содержатся определения функций **pthread_kill()** и **pthread_sigmask()**.
/usr/include/sys/types.h	Файл заголовка C/C++, содержащий определения некоторых объектов pthread.
/usr/lib/libpthreads.a	32/64-разрядная библиотека объектов pthread для стандартов UNIX98 и POSIX 1003.1c.
/usr/lib/libpthreads_compat.a	32-разрядная библиотека объектов pthread для проекта 7 стандарта POSIX 1003.1c.
/usr/lib/profiled/libpthreads.a	Оптимизированная 32/64-разрядная библиотека объектов pthread для стандартов UNIX98 и POSIX 1003.1c.
/usr/lib/profiled/libpthreads_compat.a	Оптимизированная 32-разрядная библиотека объектов pthread для проекта 7 стандарта POSIX 1003.1c.

Защита нитей и библиотеки поддержки нитей в AIX

В этом разделе рассмотрены библиотеки поддержки нитей в AIX.

Теперь все приложения по умолчанию считаются приложениями с несколькими нитями, хотя в действительности у большинства из них нить только одна. Библиотеки с защитой нитей перечислены ниже:

Библиотеки с поддержкой нитей		
libbsd.a	libc.a	libm.a
libsvid.a	libtli.a	libxti.a
libnetsvc.a		

Библиотеки нитей POSIX

Существуют следующие библиотеки нитей POSIX:

Библиотека нитей POSIX libpthreads.a

Библиотека нитей **libpthreads.a** основана на промышленном стандарте POSIX 1003.1c для переносимых API пользовательских нитей. Любая программа, рассчитанная на работу с библиотекой нитей POSIX, будет правильно работать с другой библиотекой нитей POSIX; от реализации зависят только производительность программы и две-три функции в библиотеке нитей. Для повышения уровня переносимости библиотек нитей реализация некоторых средств

программирования в стандарте POSIX объявлена необязательной. Более подробная информация об опциях POSIX приведена в разделе Необязательные компоненты библиотеки работы с нитями.

Библиотека нитей POSIX проекта 7 libpthreads_compat.a

В AIX обеспечивается двоичная совместимость с ранее созданными приложениями с несколькими нитями, соответствующими проекту 7 стандарта нитей POSIX. Эти приложения правильно работают без повторной компоновки. Библиотека **libpthreads_compat.a** нужна только для совместимости с более ранними версиями приложений, написанных по черновой версии стандарта POSIX (POSIX Thread Standard, Draft 7). Во всех новых приложениях должна использоваться библиотека **libpthreads.a**, поддерживающая как 32-разрядные, так и 64-разрядные приложения. Библиотека **libpthreads_compat.a** поддерживает только 32-разрядные приложения. Начиная с AIX 5.1 библиотека **libpthreads.a** поддерживает Single UNIX Specification версии 2, которая включает окончательную версию стандарта POSIX 1003.1c Pthread Standard.

Понятия, связанные с данным:

“Достоинства нитей” на стр. 498
Производительность программ с несколькими нитями выше, чем у обычных параллельных программ, использующих несколько процессов. В многопроцессорных системах нити дают дополнительный выигрыш в производительности.

“Необязательные компоненты библиотеки работы с нитями” на стр. 470
В данном разделе описаны расширенные атрибуты нитей, взаимных блокировок и переменных условий.

“Создание программ с несколькими нитями” на стр. 485
Создание программ с несколькими нитями аналогично разработке программ, состоящих из нескольких процессов. Процесс разработки программы включает в себя компиляцию и отладку исходного кода.

Создание нитей

Создание нитей отличается от создания процессов тем, что между нитями не существует "родственных" отношений (предок-потомок).

Все нити, за исключением *главной нити*, автоматически создаваемой при создании процесса, находятся на одном и том же уровне иерархии. Для нити не ведется список порожденных нитей; кроме того, нить не знает, какая нить ее породила.

При создании нити необходимо указать процедуру точки входа и аргумент. Каждой нити соответствует процедура точки входа с одним аргументом. Несколько нитей могут работать с одной и той же процедурой точки входа.

Нити соответствует набор атрибутов, которые определяют ее характеристики. Для установки атрибутов перед созданием нити необходимо определить объект атрибутов нити.

Объект атрибутов нити

Атрибуты нити хранятся в объекте со скрытой реализацией - *объекте атрибутов нити*, который используется при создании нити. Этот объект хранит набор атрибутов, зависящий от реализации опций POSIX. Обращение к объекту осуществляется с помощью переменной типа **pthread_attr_t**. В AIX тип данных **pthread_attr_t** обозначает указатель; в других системах это может быть структура или другой тип данных.

Создание и удаление объектов атрибутов нити

o

Объект атрибутов нити инициализируется значениями по умолчанию с помощью процедуры **pthread_attr_init**. Для работы с атрибутами предназначены специальные функции. Объект атрибутов нити удаляется с помощью функции **pthread_attr_destroy**. В зависимости от реализации библиотеки нитей, эта функция может освобождать память, динамически захваченную функцией **pthread_attr_init**.

В приведенном ниже примере объект атрибутов нити создается и инициализируется со значениями по умолчанию, затем используется при создании нити и удаляется:

```
pthread_attr_t attributes;
                /* создается объект атрибутов */
...
if (!pthread_attr_init(&attributes)) {
                /* объект атрибутов инициализируется */
        ...
                /* работа с объектом атрибутов */
        ...
        pthread_attr_destroy(&attributes);
                /* удаление объекта атрибутов */
}
```

Один объект атрибутов может применяться для создания нескольких нитей. Между вызовами операции создания нитей атрибуты объекта можно изменить. После создания нитей объект атрибутов можно удалить - эта операция не повлияет на работу созданных нитей.

Атрибут detachstate

Следующий атрибут определен всегда:

Detachstate
Определяет состояние запуска нити.

Значение атрибута можно считать процедурой **pthread_attr_getdetachstate** и установить процедурой **pthread_attr_setdetachstate**. Атрибут может принимать значение одной из следующих констант:

PTHREAD_CREATE_DETACHED
Указывает, что будет создана автономная нить

PTHREAD_CREATE_JOINABLE
Указывает, что будет создана подключаемая нить

Значение по умолчанию - **PTHREAD_CREATE_JOINABLE**.

Если была создана подключаемая нить, для нее необходимо вызвать функцию **pthread_join**. В противном случае в системе может оказаться недостаточно памяти для создания новой нити, так как каждая нить занимает относительно большой объем. Более подробная информация о функции pthread_join приведена в разделе Вызов функции pthread_join.

Прочие атрибуты нитей

В AIX дополнительно определены некоторые атрибуты, предназначенные для более точного управления нитями. Для их применения могут потребоваться специальные права доступа. Большинство программ правильно работают в том случае, если этим атрибутам присвоены значения по умолчанию. Применение перечисленных ниже атрибутов описано в разделе Работа с атрибутом inheritsched.

Область действия
Задает область действия нити

Inheritsched
Задает параметры наследования атрибутов планировщика нитей

Schedparam
Задает параметры планировщика для нити

Schedpolicy
Задает стратегию планирования для нити

Применение перечисленных ниже атрибутов стека описано в разделе Атрибуты стека.

Stacksize
Задает размер стека нити

Stackaddr
Задает адрес стека нити

Guardsize
Задает размер контрольной области стека нити

Создание нити с помощью функции pthread_create

Для создания нити предназначена функция **pthread_create**. Эта функция создает новую нить и запускает ее.

Применение объекта атрибутов нити

В вызове процедуры **pthread_create** можно указать объект атрибутов нити. Если задан указатель **NULL**, нить создается со значением атрибутов по умолчанию. Следовательно, следующий фрагмент кода:

```
pthread_t thread;
pthread_attr_t attr;
...
pthread_attr_init(&attr);
pthread_create(&thread, &attr, init_routine, NULL);
pthread_attr_destroy(&attr);
```

эквивалентен следующему:

```
pthread_t thread;
...
pthread_create(&thread, NULL, init_routine, NULL);
```

Процедура точки входа

В вызове процедуры **pthread_create** должна указываться процедура точки входа. Эта процедура, описанная в программе, аналогична функции **main** процесса. Она является первой процедурой, запускаемой в новой нити. При выходе из этой процедуры нить автоматически завершается.

Процедура точки входа имеет один параметр - указатель типа void, который задается при вызове **pthread_create**. Он может применяться для передачи указателя на некоторые данные - например, на строку или структуру. Создающая (вызывающая процедуру **pthread_create**) и создаваемая нить должны согласовать фактический тип этого указателя.

Процедура точки входа возвращает указатель типа void. После завершения нити этот указатель хранится библиотекой нитей до момента удаления нити. За дополнительной информацией о применении этого указателя обратитесь к разделу Возврат информации из нити.

Возвращаемая информация

Процедура **pthread_create** возвращает идентификатор новой нити. Вызывающая нить может использовать этот идентификатор для выполнения различных действий с созданной нитью.

В зависимости от заданных для нитей параметров планирования, новая нить может начать работу до того, как процедура **pthread_create** вернет значение вызывающей программе. Может оказаться, что в момент завершения работы функции **pthread_create** новая нить уже будет завершена. В этом случае процедура **pthread_create** возвращает в параметре *thread* неправильный идентификатор. Поэтому при работе с процедурами библиотеки нитей, которым передается идентификатор нити в качестве параметра, необходимо сравнивать результаты операций с кодом ошибки **ESRCH**.

Если процедура **pthread_create** не может создать нить, то параметр *thread* содержит недопустимый идентификатор, а процедура возвращает код ошибки. Дополнительная информация приведена в разделе Пример программы с несколькими нитями.

Работа с идентификаторами нитей

Идентификатор новой нити передается вызывающей нити в параметре *thread*. Идентификатор текущей нити можно получить с помощью процедуры **pthread_self**.

ИД нити - это объект со скрытой реализацией типа **pthread_t**. В AIX тип данных **pthread_t** представляет собой целое число (integer). В других системах он может быть структурой, указателем или другим типом.

Для повышения переносимости программ, использующих библиотеку нитей, ИД нити должен всегда обрабатываться как объект со скрытой реализацией. По этой причине сравнение идентификаторов нитей должно выполняться с помощью процедуры **pthread_equal**. Не используйте оператор сравнения C (**==**), так как тип данных **pthread_t** может отличаться от арифметического типа данных или указателя.

Понятия, связанные с данным:

“Создание программ с несколькими нитями” на стр. 485
Создание программ с несколькими нитями аналогично разработке программ, состоящих из нескольких процессов. Процесс разработки программы включает в себя компиляцию и отладку исходного кода.

Завершение работы нитей

Нить автоматически завершает работу при возврате из процедуры точки входа.

Кроме того, нить может явно завершить свою работу или работу любой другой нити в рамках процесса. Такая процедура называется *принудительным завершением*. Поскольку все нити работают с общей областью данных, каждая нить при завершении работы должна выполнять очистку; для этой цели в библиотеке нитей предусмотрены процедуры очистки.

Выход из нити

Процесс может завершить свою работу в любой момент и из любой нити с помощью процедуры **exit**. Аналогично, нить может завершить свою работу в любой момент вызовом процедуры **pthread_exit**.

Вызов процедуры **exit** приводит к завершению работы процесса в целом, включая все его нити. В программах с несколькими нитями процедуру **exit** следует вызывать только в том случае, когда необходимо завершить весь процесс: например, при возникновении неисправимой ошибки. Вместо этой процедуры следует пользоваться процедурой **pthread_exit**, в том числе и для завершения главной нити.

Вызов процедуры **pthread_exit** приводит к завершению нити, вызвавшей эту процедуру. Параметр *status* при этом сохраняется библиотекой; позднее этот параметр можно использовать при стыковке с завершенной нитью. Вызов процедуры **pthread_exit** похож на возврат из главной процедуры нити, с некоторыми отличиями. Результат возврата из главной процедуры нити зависит от типа нити:

- При возврате из главной нити неявно вызывается процедура **exit**, т.е. завершается обработка всех нитей процесса.
- При возврате из нити, не являющейся главной, неявно вызывается процедура **pthread_exit**. Возвращаемое значение играет ту же роль, что и параметр *status* процедуры **pthread_exit**.

Для предотвращения неявного вызова функции **exit**, рекомендуется всегда завершать работу нити вызовом **pthread_exit**.

Естественное завершение главной нити (например, путем вызова процедуры **pthread_exit** из процедуры **main**) не приводит к завершению процесса. Завершается только главная нить. Если главная нить завершила работу, то процесс завершится при завершении работы его последней нити. В этом случае процесс возвращает код 0.

Следующая программа показывает ровно 10 сообщений на каждом языке. Для этого после создания двух нитей в процедуре **main** вызывается процедура **pthread_exit**, а в процедуре **Thread** предусмотрен цикл.

```
#include <pthread.h>    /* первый включаемый файл - pthread.h */
#include <stdio.h>      /* поддержка функции printf() */

void *Thread(void *string)

{
        int i;

        for (i=0; i<10; i++)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_exit(NULL);
}
```

Функция **pthread_exit** освобождает всю память, занятую данными нити, включая стек нити. Любые данные, помещенные в стек, становятся неопределенными, так как стек освобожден и соответствующая область памяти может использоваться другой нитью. Следовательно, необходимо уничтожить объекты, управляющие синхронизацией нитей (взаимные блокировки и переменные условия) и размещенные в стеке нити, до вызова процедуры **pthread_exit** для этой нити.

В отличие от функции **exit**, функция **pthread_exit** не очищает ресурсы системы, общие для всех нитей. Например, процедура **pthread_exit** не закрывает открытые файлы, так как с ними могут работать другие нити.

Принудительное завершение работы нити

Механизм принудительного завершения работы позволяет нити завершить любую другую нить процесса в управляемом режиме. Целевая нить (т.е. нить, которую следует завершить) может несколькими способами отложить запрос на завершение, выполнив сначала необходимую процедуру очистки. При принудительном завершении работы нити она неявно вызывает процедуру **pthread_exit((void *)-1)**.

Для генерации запроса на принудительное завершение нити вызывается процедура **pthread_cancel**. При возврате из этой процедуры запрос регистрируется, но целевая нить может еще выполняться. Вызов процедуры **pthread_cancel** заканчивается неудачей только в случае, если указан неверный ИД нити.

Запрет и режим принудительного завершения

Запрет и режим принудительного завершения определяют действия, которые будут выполнены при получении запроса на принудительное завершение. Каждая нить управляет своими параметрами запрета и режима принудительного завершения с помощью процедур **pthread_setcancelstate** и **pthread_setcanceltype**.

Существует три варианта значений параметра запрета принудительного завершения и режима принудительного завершения, как показано в следующей таблице.

Запрет завершения	**Режим завершения**	**Вариант**
Запрещено	Любой (параметр игнорируется)	Принудительное завершение запрещено
Разрешено	Отложенное	Отложенное принудительное завершение
Разрешено	Асинхронный кабель	Асинхронное принудительное завершение

Ниже описаны возможные варианты:

- *Принудительное завершение запрещено*. Любой запрос на принудительное завершение переводится в ожидающее состояние до тех пор, пока принудительное завершение не будет разрешено, или пока обработка нити не завершится другим путем.

  Запрет принудительного завершения следует устанавливать только в случае, если операцию, выполняемую нитью, нельзя прерывать. Например, если нить выполняет несколько сложных операций сохранения файлов (например, сохранение файлов индексированной базы данных), то принудительное завершение ее обработки может привести к нарушению целостности данных. Поэтому следует запретить принудительное завершение этой нити на время сохранения файлов.
- *Отложенное принудительное завершение*. Любой запрос на принудительное завершение переводится в ожидающее состояние до тех пор, пока нить не достигнет следующей точки завершения. Это состояние установлено по умолчанию.

  Такое сочетание параметров обеспечивает принудительное завершение нити, но только в определенные моменты ее выполнения, называемые *точками завершения*. При принудительном завершении нити в точке завершения система остается в согласованном состоянии, однако возможно нарушение целостности пользовательских данных или сохранение блокировок, установленных завершенной нитью. Избежать подобных ошибок можно путем применения процедур очистки или запрета принудительного завершения на соответствующих этапах выполнения. Дополнительная информация приведена в разделе Процедуры очистки .
- *Асинхронное принудительное завершение*. Запросы на принудительное завершение выполняются немедленно.

  Асинхронное завершение нити, работающей с ресурсами, может вызвать переход процесса или системы в целом в такое состояние, что восстановление будет крайне трудным или даже невозможным. Дополнительная информация о безопасном асинхронном завершении нити приведена в разделе Поддержка асинхронного завершения.

Асинхронное завершение, поддержка

Функция называется *устойчивой к асинхронному завершению*, если ее асинхронное завершение на любой инструкции не приводит к повреждению каких-либо ресурсов.

Функции, работающие с ресурсами, не могут быть устойчивыми к асинхронному завершению. Например, при вызове процедуры **malloc** без запрета асинхронного завершения она может успешно захватить ресурс, но, если в момент возврата из процедуры поступит запрос на принудительное завершение, программа не сможет определить, был ли ресурс получен.

По изложенным выше причинам большинство библиотечных процедур не являются устойчивыми к асинхронному завершению. Асинхронное принудительное завершение рекомендуется применять только в том случае, когда точно известно, что выполняемая операция не захватывает ресурсы, а вызываемые библиотечные функции устойчивы к асинхронному завершению.

Следующие функции устойчивы к асинхронному завершению: при их принудительном завершении, даже асинхронном, ресурсы будут правильно освобождены.

- **pthread_cancel**
- **pthread_setcancelstate**
- **pthread_setcanceltype**

Вместо асинхронного завершения можно применять отложенное завершение, явно указав точки завершения с помощью функции **pthread_testcancel**.

Точки завершения

Точки завершения - это точки внутри той или иной процедуры, в которых при включенном режиме отложенного завершения нить должна выполнить все ожидающие запросы на принудительное завершение. Любая из этих процедур может блокировать вызывающую нить или выполняться неограниченное время.

Явную точку завершения можно создать с помощью функции **pthread_testcancel**. Эта функция просто создает точку завершения. Если разрешено отложенное завершение, то при наличии ожидающих запросов на принудительное завершение в этой точке они будут выполнены, и нить завершится. В противном случае обработка нити будет продолжена.

Следующие функции также создают точки завершения:

- **pthread_cond_wait**
- **pthread_cond_timedwait**
- **pthread_join**

Функции **pthread_mutex_lock** и **pthread_mutex_trylock** не предоставляют точку завершения. В противном случае неявные точки завершения создавались бы во всех функциях, вызывающих данные функции. Число точек завершения было бы непомерно велико, что сильно затруднило бы процесс программирования. Нужно было бы либо многократно запрещать и разрешать принудительное завершение, либо везде добавлять процедуры очистки. За дополнительной информацией об этих функциях обратитесь к разделу Использование взаимных блокировок.

Точки завершения создаются при вызове следующих функций внутри нити:

Функция	
aio_suspend	close
creat	fcntl
fsync	getmsg
getpmsg	lockf
mq_receive	mq_send
msgrcv	msgsnd
msync	nanosleep
open	pause
poll	pread
pthread_cond_timedwait	pthread_cond_wait
pthread_join	pthread_testcancel
putpmsg	pwrite
read	readv
select	sem_wait
sigpause	sigsuspend
sigtimedwait	sigwait
sigwaitinfo	sleep
system	tcdrain
usleep	wait
wait3	waitid
waitpid	запись

Функция
writev

Кроме того, точки завершения могут создаваться при вызове следующих функций:

Функция		
catclose	catgets	catopen
closedir	closelog	ctermid
dbm_close	dbm_delete	dbm_fetch
dbm_nextkey	dbm_open	dbm_store
dlclose	dlopen	endgrent
endpwent	fwprintf	fwrite
fwscanf	getc	getc_unlocked
getchar	getchar_unlocked	getcwd
getdate	getgrent	getgrgid
getgrgid_r	getgrnam	getgrnam_r
getlogin	getlogin_r	popen
printf	putc	putc_unlocked
putchar	putchar_unlocked	puts
pututxline	putw	putwc
putwchar	readdir	readdir_r
remove	rename	rewind
endutxent	fclose	fcntl
fflush	fgetc	fgetpos
fgets	fgetwc	fgetws
fopen	fprintf	fputc
fputs	getpwent	getpwnam
getpwnam_r	getpwuid	getpwuid_r
gets	getutxent	getutxid
getutxline	getw	getwc
getwchar	getwd	rewinddir
scanf	seekdir	semop
setgrent	setpwent	setutxent
strerror	syslog	tmpfile
tmpnam	ttyname	ttyname_r
fputwc	fputws	fread
freopen	fscanf	fseek
fseeko	fsetpos	ftell
ftello	ftw	glob
iconv_close	iconv_open	ioctl
lseek	mkstemp	nftw
opendir	openlog	pclose
perror	ungetc	ungetwc
unlink	vfprintf	vfwprintf
vprintf	vwprintf	wprintf
wscanf		

Побочные эффекты выполнения запроса на принудительное завершение приостановленной нити те же, что и в случае, когда выполнение обычной программы с одной нитью прерывается по сигналу, и данная функция возвращает [EINTR]. Все подобные побочные эффекты проявляются до вызова процедур очистки.

Если для нити разрешено принудительное завершение и поступил запрос на ее принудительное разрешение, а нить вызвала функцию **pthread_testcancel**, то запрос выполняется до возврата из функции **pthread_testcancel**. Если для нити разрешено принудительное завершение, и в тот момент, когда нить приостановлена в точке завершения до наступления какого-либо события, поступил запрос на асинхронное завершение, этот запрос выполняется. Однако в случае, если выполнение нити приостановлено в точке завершения, а ожидаемое событие произошло до выполнения запроса на принудительное завершение, то, в

зависимости от порядка событий, либо нить будет принудительно завершена, либо запрос останется в состоянии ожидания, и выполнение нити продолжится в обычном режиме.

Пример принудительного завершения

В следующем примере обе нити, записывающие сообщения, принудительно завершаются через 10 секунд после начала выполнения, если каждая из них напечатала свое сообщение хотя бы пять раз.

```
#include <pthread.h>    /* первый включаемый файл - pthread.h */
#include <stdio.h>      /* поддержка функции printf() */
#include <unistd.h>     /* поддержка функции sleep()          */

void *Thread(void *string)
{
        int i;
        int o_state;

        /* запрет принудительного завершения */
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &o_state);

        /* записывает пять сообщений */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);

        /* разрешение принудительного завершения */
        pthread_setcancelstate(o_state, &o_state);

        /* продолжение записи */
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        /* создание обеих нитей */
        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                return -1;
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                return -1;
        /* ожидание */
        sleep(10);

        /* запрос на принудительное завершение */
        pthread_cancel(e_th);
        pthread_cancel(f_th);

        /* ожидание */
        sleep(10);
        pthread_exit(NULL);
}
```

Функции таймера и перехода в состояние ожидания

Функции таймера выполняются в контексте вызывающей нити. Следовательно, при срабатывании таймера контрольная функция вызывается в нити. При переходе процесса или нити в состояние ожидания этот процесс или нить освобождает ресурсы процессора. В процессах с несколькими нитями перевести в состояние ожидания можно только вызывающую нить.

Процедуры очистки

Процедуры очистки предоставляют переносимый на другие платформы механизм освобождения ресурсов и восстановления исходных значений при завершении нити.

Вызов процедур очистки

Процедуры очистки создаются для каждой нити отдельно. Для нити можно создать несколько процедур очистки. Они хранятся в стеке LIFO нити. Ниже перечислены случаи, в которых вызываются все процедуры очистки:

- При возврате из процедуры точки входа для данной нити.
- При вызове функции **pthread_exit** из нити.
- При обработке запроса на принудительное завершение.

Для добавления процедуры очистки в стек очистки служит функция **pthread_cleanup_push**. Функция **pthread_cleanup_pop** извлекает из стека самую верхнюю процедуру очистки и, при необходимости, выполняет ее. Эту процедуру применяют также для удаления ненужных процедур очистки.

Процедура очистки представляет собой пользовательскую процедуру с одним параметром типа "указатель на void", передаваемым при вызове процедуры **pthread_cleanup_push**. Это может быть указатель на данные, которые необходимы процедуре очистки.

В приведенном ниже примере для выполнения некоторой операции был выделен буфер. Если разрешено отложенное принудительное завершение, то эта операция может быть прервана в любой точке завершения. В этом случае процедура очистки должна освободить память, занятую буфером.

```
/* процедура очистки */

cleaner(void *buffer)

{
        free(buffer);
}

/* фрагмент другой процедуры */
...
myBuf = malloc(1000);
if (myBuf != NULL) {

        pthread_cleanup_push(cleaner, myBuf);

        /*
         *       выполняются любые операции с буфером,
         *       в том числе вызовы других функций
         *       и точек завершения
         */

        /* выталкивание процедуры очистки и освобождение буфера за 1 вызов */
        pthread_cleanup_pop(1);
}
```

Поскольку установлен режим отложенного завершения, нить не будет выполнять запросы на принудительное завершение с момента выделения буфера до регистрации процедуры очистки, поскольку ни в процедуре **malloc**, ни в процедуре **pthread_cleanup_push** нет точек завершения. При извлечении процедуры очистки из стека она выполняется и освобождает буфер. В более сложных программах при извлечении

процедуры очистки из стека ее можно и не выполнять, так как эти процедуры следует применять в качестве средств обработки аварийных ситуаций для защищенного фрагмента кода.

Согласование операций помещения в стек и выталкивания из стека

Процедуры **pthread_cleanup_push** и **pthread_cleanup_pop** должны указываться в каждой лексической области попарно, т.е. в пределах одной функции и одного блока операторов должно быть одинаковое число процедур того и другого типа. Эти процедуры можно рассматривать как открывающие и закрывающие скобки, в которые заключен защищенный фрагмент кода.

Это правило обусловлено тем обстоятельством, что в некоторых системах данные процедуры реализованы в виде макрокоманд. Процедура **pthread_cleanup_push** реализована в виде открывающей скобки, за которой следует последовательность операторов:

```
#define pthread_cleanup_push(rtm,arg) { \
         /* прочие операторы */
```

Процедура **pthread_cleanup_pop** реализована в виде закрывающей скобки, за которой следуют прочие операторы:

```
#define pthread_cleanup_pop(ex) \
         /* другие операторы */
}
```

Для того чтобы избежать ошибок на стадии компиляции или неправильной работы программы после ее переноса на другую платформу, следует соблюдать правила соответствия вызовов процедур **pthread_cleanup_push** и **pthread_cleanup_pop**.

В AIX **pthread_cleanup_push** и **pthread_cleanup_pop** реализованы в виде библиотечных функций, поэтому соблюдать правило соответствия вызовов в пределах блока операторов не требуется. Однако его необходимо соблюдать в рамках программы, поскольку процедуры очистки помещаются в стек.

Функция	Описание
pthread_attr_destroy	Удаляет объект атрибутов нити.
pthread_attr_getdetachstate	Возвращает значение атрибута detachstate объекта атрибутов нити.
pthread_attr_init	Создает объект атрибутов нити и инициализирует его значениями по умолчанию.
pthread_cancel	Отправляет запрос на принудительное завершение нити.
pthread_cleanup_pop	Удаляет (и дополнительно выполняет) процедуру с вершины стека очистки вызывающей нити.
pthread_cleanup_push	Помещает процедуру в стек очистки вызывающей нити.
pthread_create	Создает новую нить, инициализирует ее атрибуты и подготавливает ее к запуску.
pthread_equal	Сравнивает идентификаторы двух нитей.
pthread_exit	Завершает вызывающую нить.
pthread_self	Возвращает идентификатор текущей нити.
pthread_setcancelstate	Устанавливает состояние завершения текущей нити.
pthread_setcanceltype	Устанавливает тип завершения текущей нити.
pthread_testcancel	Создает точку завершения в вызывающей нити.

Понятия, связанные с данным:

“Разовая инициализация” на стр. 455
Некоторые библиотеки на языке C используют динамическую инициализацию, когда глобальная инициализация библиотеки выполняется при первом вызове процедуры из этой библиотеки.

Обзор синхронизации

Одно из главных преимуществ нитей заключается в простоте средств синхронизации.

Синхронизация нитей необходима для их эффективного взаимодействия. Предусмотрены следующие средства синхронизации:

- Неявный обмен данными путем изменения общих данных
- Явный обмен данными путем уведомления других нитей о возникших событиях

Более сложные объекты синхронизации можно создать на основе этих базовых объектов.

В библиотеке нитей предусмотрены следующие механизмы синхронизации. На основе этих простых, но мощных механизмов можно создать более сложные средства синхронизации.

В библиотеке нитей предусмотрены следующие механизмы синхронизации:

- Взаимные блокировки (см. раздел Использование взаимных блокировок)
- Переменные условия (см. раздел Использование переменных условия).
- Блокировки чтения-записи (см. раздел Применение блокировок чтения-записи)
- Стыковки с нитями (см. раздел Стыковка с нитями).

На основе этих простых, но мощных механизмов можно создать более сложные средства синхронизации.

Понятия, связанные с данным:

“Создание сложных объектов синхронизации” на стр. 460
С помощью функций из библиотеки работы с нитями можно создавать более сложные объекты синхронизации.

Использование взаимных блокировок

Взаимная блокировка - это взаимоисключающая блокировка. Захватить такую блокировку может только одна нить.

Взаимные блокировки применяются для предотвращения одновременного доступа к общим данным. У взаимной блокировки есть атрибуты, которые определяют ее свойства.

Объект атрибутов взаимной блокировки

Как и нити, взаимные блокировки создаются с помощью объектов атрибутов. Объект атрибутов взаимной блокировки представляет собой абстрактный объект, содержащий несколько атрибутов, набор которых зависит от реализации компонентов POSIX. It is accessed through a variable of type **pthread_mutexattr_t**. В AIX тип данных **pthread_mutexattr_t** обозначает указатель; в других системах это может быть структура или другой тип данных.

Создание и удаление объекта атрибутов взаимной блокировки

Объект атрибутов взаимной блокировки можно инициализировать значениями по умолчанию с помощью функции **pthread_mutexattr_init**. Для работы с атрибутами предназначены специальные функции. Для удаления объекта атрибутов взаимной блокировки предназначена функция **pthread_mutexattr_destroy**. В некоторых реализациях библиотеки нитей эта функция освобождает память, динамически выделенную функцией **pthread_mutexattr_init**.

В приведенном ниже примере объект атрибутов создается и инициализируется значениями по умолчанию. Когда объект становится ненужным, он уничтожается.

```
pthread_mutexattr_t attributes;
                /* создается объект атрибутов */
...
if (!pthread_mutexattr_init(&attributes)) {
                /* объект атрибутов инициализируется */
        ...
                /* работа с объектом атрибутов */
```

```
        ...
        pthread_mutexattr_destroy(&attributes);
                /* удаление объекта атрибутов */
}
```

С помощью одного объекта атрибутов можно создать несколько взаимных блокировок. Этот объект можно изменить в промежутке между созданием двух взаимных блокировок. После создания взаимных блокировок объект атрибутов можно удалить - это никак не отразится на созданных взаимных блокировках.

Атрибуты взаимной блокировки

Определены следующие атрибуты взаимной блокировки:

Атрибут	Описание
Протокол	Задает протокол, защищающий приоритеты нитей от изменения после установки взаимной блокировки. Этот атрибут зависит от опции наследования приоритетов или опции защиты приоритетов POSIX.
Process-shared	Задает режим совместного использования взаимной блокировки. Этот атрибут зависит от значения опции совместного использования POSIX.

Дополнительная информация об этих атрибутах приведена в разделах Необязательные компоненты библиотеки работы с нитями и Планирование синхронизации.

Создание и удаление взаимных блокировок

Взаимная блокировка создается вызовом функции **pthread_mutex_init**. При вызове этой функции можно указать объект атрибутов взаимной блокировки. Если значение указателя будет равно **NULL**, атрибутам будут присвоены значения по умолчанию. Следовательно, следующий фрагмент кода:

```
pthread_mutex_t mutex;
pthread_mutexattr_t    attr;
...
pthread_mutexattr_init(&attr);
pthread_mutex_init(&mutex, &attr);
pthread_mutexattr_destroy(&attr);
```

эквивалентен следующему:

```
pthread_mutex_t mutex;
...
pthread_mutex_init(&mutex, NULL);
```

ИД созданной взаимной блокировки возвращается нити через параметр *mutex*. ИД взаимной блокировки представляет собой прозрачный объект типа **pthread_mutex_t**. В AIX тип данных **pthread_mutex_t** обозначает структуру; в других системах это может быть указатель или другой тип данных.

Взаимная блокировка должна создаваться один раз. Однако вызывать функцию **pthread_mutex_init** более одного раза с одним и тем же параметром *mutex* (например, в двух разных нитях, одновременно выполняющих один и тот же фрагмент кода) нельзя. Предотвратить повторное создание взаимной блокировки можно следующими способами:

- Путем вызова функции **pthread_mutex_init** перед созданием остальных нитей, которые будут применять эту взаимную блокировку, например, в главной нити.
- Путем вызова функции **pthread_mutex_init** в процедуре однократной инициализации. Дополнительная информация приведена в разделе Разовая инициализация.
- С помощью статических взаимных блокировок, инициализируемых макрокомандой **PTHREAD_MUTEX_INITIALIZER**; при этом взаимной блокировке присваиваются атрибуты по умолчанию.

После того как взаимная блокировка станет не нужна, удалите ее с помощью функции **pthread_mutex_destroy**. Эта функция освобождает все области памяти, выделенные функцией **pthread_mutex_init**. После удаления взаимной блокировки ту же самую переменную **pthread_mutex_t** можно вновь использовать для создания другой взаимной блокировки. Например, приведенный ниже фрагмент кода, хотя и несколько искусственный, не содержит ошибок:

```
pthread_mutex_t mutex;
...
for (i = 0; i < 10; i++) {

        /* создание взаимной блокировки */
        pthread_mutex_init(&mutex, NULL);

        /* применение взаимной блокировки */

        /* удаление взаимной блокировки */
        pthread_mutex_destroy(&mutex);
}
```

Как и другие общие системные ресурсы, взаимные блокировки, размещенные в стеке нити, необходимо удалить до завершения нити. В библиотеке нитей хранится скомпонованный список взаимных блокировок. Следовательно, при освобождении стека, содержащего взаимную блокировку, этот список будет поврежден.

Типы взаимных блокировок

Тип взаимной блокировки определяет алгоритм ее работы. Предусмотрены следующие типы взаимных блокировок:

PTHREAD_MUTEX_DEFAULT или PTHREAD_MUTEX_NORMAL
: Если нить попытается повторно захватить такую взаимную блокировку с помощью функции **pthread_mutex_lock**, предварительно не освободив ее, то возникнет тупик. Этот тип применяется по умолчанию.

PTHREAD_MUTEX_ERRORCHECK
: Предотвращает возникновение тупиков, возвращая ненулевое значение в том случае, если нить пытается повторно захватить блокировку, предварительно не освободив ее.

PTHREAD_MUTEX_RECURSIVE
: Нить может рекурсивно захватывать взаимную блокировку этого типа с помощью функции **pthread_mutex_lock**. При этом тупиковые ситуации не возникают, а функция **pthread_mutex_lock** не возвращает ненулевое значение. Для того чтобы освободить взаимную блокировку, нить должна вызвать функцию **pthread_mutex_unlock** столько же раз, сколько была вызвана функция **pthread_mutex_lock**.

Все взаимные блокировки создаются с типом **PTHREAD_MUTEX_NORMAL**. После создания блокировки ее тип можно изменить с помощью API **pthread_mutexattr_settype**.

Ниже приведен пример создания и применения рекурсивной взаимной блокировки:

```
pthread_mutexattr_t    attr;
pthread_mutex_t mutex;

pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex, &attr);

struct {
        int a;
        int b;
        int c;
} A;

f()
{
```

```
        pthread_mutex_lock(&mutex);
        A.a++;
        g();
        A.c = 0;
        pthread_mutex_unlock(&mutex);
}

g()
{
        pthread_mutex_lock(&mutex);
        A.b += A.a;
        pthread_mutex_unlock(&mutex);
}
```

Захват и освобождение взаимных блокировок

Взаимная блокировка - это обычная блокировка, у которой есть два состояния: занята и свободна. После создания взаимная блокировка свободна. Функция **pthread_mutex_lock** захватывает взаимную блокировку на следующих условиях:

- Если взаимная блокировка свободна, функция ее захватывает.
- Если блокировка уже захвачена другой нитью, функция блокирует вызывающую нить до освобождения взаимной блокировки.
- Если взаимная блокировка уже захвачена вызывающей нитью, то возникнет тупик, либо функция вернет сообщение об ошибке, в зависимости от типа взаимной блокировки.

Функция **pthread_mutex_trylock** работает так же, как и функция **pthread_mutex_lock**, но не блокирует вызывающую нить:

- Если взаимная блокировка свободна, функция ее захватывает.
- Если взаимная блокировка захвачена, функция возвращает сообщение об ошибке.

Нить, захватившую взаимную блокировку, часто называют *владельцем* блокировки.

Функция **pthread_mutex_unlock** освобождает взаимную блокировку, если ее владельцем является вызывающая нить, соблюдая следующие условия:

- Если взаимная блокировка уже свободна, функция возвращает сообщение об ошибке.
- Если владельцем взаимной блокировки является вызывающая нить, функция освобождает блокировку.
- Если владельцем взаимной блокировки является другая нить, функция возвращает сообщение об ошибке или освобождает блокировку, в зависимости от типа взаимной блокировки. Освобождать блокировку не рекомендуется, так как блокировки, как правило, захватываются и освобождаются одной и той же нитью.

Поскольку операцию захвата блокировки отменить нельзя, нить, ожидающую освобождения взаимной блокировки, нельзя принудительно завершить. В связи с этим рекомендуется устанавливать взаимные блокировки на короткое время, например, для защиты общих данных. Дополнительная информация приведена в разделах Точки завершения и Принудительное завершение работы нити.

Защита данных с помощью взаимных блокировок

Взаимные блокировки либо устанавливаются для защиты данных, либо служат основой для создания более сложных объектов синхронизации. Дополнительная информация о реализации длительных блокировок и блокировок чтения и записи с приоритетом записи приведена в разделе “Использование взаимных блокировок” на стр. 424.

Пример применения взаимной блокировки

Взаимные блокировки позволяют защитить данные от параллельного изменения несколькими нитями. Например, приложение для работы с базой данных может создать несколько нитей для одновременной обработки нескольких запросов. При этом сама база данных будет защищена взаимной блокировкой **db_mutex**. Например:

```
/* главная нить */
pthread_mutex_t mutex;
int i;
...
pthread_mutex_init(&mutex, NULL);    /* создание взаимной блокировки */
for (i = 0; i < num_req; i++)        /* создание нитей в цикле */
        pthread_create(th + i, NULL, rtn, &mutex);
...                                  /* ожидание конца сеанса */
pthread_mutex_destroy(&mutex);       /* удаление взаимной блокировки */
...

/* нить, обрабатывающая запрос */
...                                  /* ожидание запроса */
pthread_mutex_lock(&db_mutex);       /* блокировка базы данных */
...                                  /* обработка запроса */
pthread_mutex_unlock(&db_mutex);     /* разблокирование базы данных */
...
```

Главная нить создает взаимную блокировку и все нити, предназначенные для обработки запросов. Взаимная блокировка передается нити через параметр функции точки входа нити. В реальной программе адрес взаимной блокировки может передаваться во вновь созданную нить в виде одного из полей сложной структуры данных.

Предотвращение тупиковых ситуаций

Существует несколько причин, по которым в приложении с несколькими нитями может возникнуть тупик. Ниже приведено несколько примеров:

- Тупик возникает при попытке нити повторно захватить блокировку типа **PTHREAD_MUTEX_NORMAL** (тип по умолчанию).
- Тупик может возникнуть при захвате взаимных блокировок в обратном порядке. Например, в приведенном ниже фрагменте кода выполнение нитей A и B может зайти в тупик.

  ```
  /* Нить A */
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);

  /* Нить B */
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  ```

- При выполнении приложения может возникнуть так называемый тупик *ресурсов*. Например:

  ```
  struct {
                  pthread_mutex_t mutex;
                  char *buf;
          } A;

  struct {
                  pthread_mutex_t mutex;
                  char *buf;
          } B;

  struct {
                  pthread_mutex_t mutex;
                  char *buf;
          } C;

  use_all_buffers()
  {
          pthread_mutex_lock(&A.mutex);
  ```

```
        /* применение буфера A */

        pthread_mutex_lock(&B.mutex);
        /* применение буфера B */

        pthread_mutex_lock(&C.mutex);
        /* применение буфера C */

        /* Выполнение необходимых операций */
        pthread_mutex_unlock(&C.mutex);
        pthread_mutex_unlock(&B.mutex);
        pthread_mutex_unlock(&A.mutex);
}

use_buffer_a()
{
        pthread_mutex_lock(&A.mutex);
        /* применение буфера A */
        pthread_mutex_unlock(&A.mutex);
}

functionB()
{
        pthread_mutex_lock(&B.mutex);
        /* применение буфера B */
        if (..условие)
        {
          use_buffer_a();
        }
        pthread_mutex_unlock(&B.mutex);
}

/* Нить A */
use_all_buffers();

/* Нить B */
functionB();
```

Это приложение содержит две нити: нить A и нить B. Сначала запускается нить B, а затем - нить A. Если нить A вызовет функцию **use_all_buffers()** и успешно захватит **A.mutex**, то при попытке заблокировать **B.mutex** возникнет тупик, так как эта блокировка уже захвачена нитью B. В процессе выполнения функции **functionB** в нити B выполняется условие, но нить A заблокирована, поэтому нить B тоже не может продолжить работу. Она пытается захватить блокировку **A.mutex**, которая уже захвачена нитью A. Это приводит к тупику.

Во избежание такой тупиковой ситуации перед выполнением операций с ресурсами нити должны захватывать все необходимые блокировки этих ресурсов. Если захватить все блокировки не удастся, то следует освободить захваченные блокировки и начать процедуру сначала.

Взаимные блокировки и "гонки"

Взаимные блокировки позволяют предотвратить нарушение целостности данных вследствие так называемой "гонки". "Гонки" часто возникают в случае, когда несколько нитей должны выполнить действия в одной и той же области памяти, причем результат вычислений зависит от порядка действий.

Предположим, что две нити (A и B) могут увеличивать значение счетчика *X*. Если значение *X* вначале было равно 1, то после его увеличения в обеих нитях оно будет равно 3. Обе нити представляют собой независимые объекты без синхронизации. Хотя оператор C X++ выглядит предельно просто и, казалось бы, должен быть атомарным, соответствующий ассемблерный код состоит из нескольких инструкций:

```
move    X, REG
inc     REG
move    REG, X
```

Если нити из предыдущего примера одновременно выполняются на двух процессорах, или если запланировано попеременное выполнение по одной инструкции из каждой нити, может произойти следующее:

1. Нить A выполняет первую инструкцию и помещает значение *X*, равное 1, в свой регистр. После этого нить B выполняет свою инструкцию и помещает значение *X*, равное 1, в свой регистр. В следующем примере показаны значения регистров и содержимое счетчика *X*.

   ```
   Регистр нити A = 1
   Регистр нити B = 1
   X          = 1
   ```

2. Нить A выполняет вторую инструкцию и увеличивает значение в своем регистре до 2. Затем нить B увеличивает значение в своем регистре до 2. В счетчик *X* не записывается никакое значение, поэтому содержимое *X* останется прежним. В следующем примере показаны значения регистров и содержимое счетчика *X*.

   ```
   Регистр нити A = 2
   Регистр нити B = 2
   X          = 1
   ```

3. Нить A записывает содержимое своего регистра (значение 2) в счетчик *X*. Нить B также записывает содержимое своего регистра (значение 2) в счетчик *X*, заменяя значение нити A. В следующем примере показаны значения регистров и содержимое счетчика *X*.

   ```
   Регистр нити A = 2
   Регистр нити B = 2
   X          = 2
   ```

В большинстве случаев нити A и B выполняют эти инструкции единым блоком, и результат будет равен 3. Как правило, нарушение целостности данных вследствие гонки трудно обнаружить, так как гонка возникает не всегда.

Для того чтобы подобные ситуации не возникали, каждая нить должна блокировать данные перед обращением к счетчику и обновлением ячейки памяти *X*. Например, если нить A установит блокировку и обновит счетчик, то в момент снятия блокировки значение счетчика будет равно 2. После этого нить B заблокирует счетчик и обновит его значение, увеличив его значение с 2 до 3.

Использование переменных условия

Переменные условия позволяют приостановить выполнение нити, пока не наступит заданное событие или не будет соблюдено некоторое условие.

В переменной условия можно задать атрибуты, определяющие характеристики условия. Обычно в программах используются следующие объекты:

- Булевская переменная, которая указывает, выполнено ли условие
- Взаимная блокировка, которая упорядочивает доступ к булевской переменной
- Переменная условия, которая ожидает соблюдения условия

Работать с переменными условия непросто, но зато они позволяют реализовать мощные и эффективные механизмы синхронизации. Дополнительная информация о реализации длительных блокировок и семафоров с помощью переменных условия приведена в разделе Создание сложных объектов синхронизации.

После завершения работы нити соответствующая память может не очищаться; это зависит от атрибутов нити. Такие нити можно соединять с другими нитями для получения доступа к оставшейся информации. Нить, к которой нужно подключить другую нить, блокируется до завершения работы подключенной нити. Механизм подключения представляет собой разновидность механизма переменных условия, причем условием является завершение работы нити.

Объект условных атрибутов

Так же как нити и взаимные блокировки, переменные условия создают с помощью объекта атрибутов. *Объект атрибутов условия* - это абстрактный объект, который содержит один атрибут или не содержит ни одного, в зависимости от варианта реализации опций POSIX. It is accessed through a variable of type **pthread_condattr_t**. В AIX тип данных **pthread_condattr_t** обозначает указатель; в других системах это может быть структура или другой тип данных.

Создание и удаление объектов атрибутов условия

Объект атрибутов условия инициализируется значениями по умолчанию с помощью функции **pthread_condattr_init**. Ряд функций предусмотрен и для работы с атрибутами. За удаление объекта атрибутов нити отвечает функция **pthread_condattr_destroy**. В зависимости от способа реализации библиотеки нитей, эта функция может освобождать память, динамически выделенную функцией **pthread_condattr_init**.

В следующем примере объект атрибутов нити создается и инициализируется значениями по умолчанию, используется, а затем удаляется:

```
pthread_condattr_t attributes;
                /* создается объект атрибутов */
...
if (!pthread_condattr_init(&attributes)) {
                /* объект атрибутов инициализируется */
        ...
                /* работа с объектом атрибутов */
        ...
        pthread_condattr_destroy(&attributes);
                        /* удаление объекта атрибутов */
}
```

Один и тот же объект атрибутов можно использовать для создания нескольких переменных условия. Кроме того, его можно изменять в период между созданием двух переменных условия. После создания переменных условия соответствующий объект атрибутов можно удалить - это не повлияет на переменные.

Атрибут условия

Поддерживаются следующие атрибуты условия:

Совместного выполнения процессов
Задает параметры совместного использования процесса переменной условия. Данный атрибут зависит от опций POSIX совместного использования процесса.

Создание и удаление переменных условия

Для создания переменной условия предназначена функция **pthread_cond_init**. Вы можете определить объект атрибутов условия. Задав указатель **NULL**, вы определите атрибуты по умолчанию. Следовательно, следующий фрагмент кода:

```
pthread_cond_t cond;
pthread_condattr_t attr;
...
pthread_condattr_init(&attr);
pthread_cond_init(&cond, &attr);
pthread_condattr_destroy(&attr);
```

эквивалентен следующему:

```
pthread_cond_t cond;
...
pthread_cond_init(&cond, NULL);
```

ИД созданной переменной возвращается в вызывающую нить через параметр *condition*. ИД условия - это объект со скрытой реализацией типа **pthread_cond_t**. В AIX тип данных **pthread_cond_t** является структурой; в других системах он может быть указателем или каким-либо другим типом данных.

Создавать переменную условия следует только один раз. Вызывать функцию **pthread_cond_init** более одного раза с одним и тем же параметром *условия* (например, в двух параллельных нитях, выполняющих одну и ту же программу) не рекомендуется. Однократное создание переменной условия можно обеспечить тремя способами:

- Вызывайте функцию **pthread_cond_init** до создания других нитей, которые будут использовать эту переменную, например, в главной нити.
- Вызывайте функцию **pthread_cond_init** из программы с однократной инициализацией. Дополнительная информация приведена в разделе Разовая инициализация.
- Используйте статическую переменную условия, которая инициализируется специальной макрокомандой **PTHREAD_COND_INITIALIZER**; в этом случае переменная условия инициализируется со стандартными атрибутами по умолчанию.

После того как переменная условия станет не нужна, ее следует удалить с помощью функции **pthread_cond_destroy**. Это команда восстанавливает всю память, захваченную функцией **pthread_cond_init**. После удаления переменной условия можно создать другое условие с помощью той же самой переменной **pthread_cond_t**. Например, приведенный ниже фрагмент кода, хотя и несколько искусственный, не содержит ошибок:

```
pthread_cond_t cond;
...
for (i = 0; i < 10; i++) {

        /* создание переменной условия */
        pthread_cond_init(&cond, NULL);

        /* применение переменной условия */

        /* удаление переменной условия */
        pthread_cond_destroy(&cond);
}
```

Так же как и любой системный ресурс, который может использоваться несколькими нитями, переменная условия должна быть удалена из стека памяти, выделенного нити, до момента завершения нити. Библиотека нитей хранит список переменных условия; таким образом, освобождение стека, в котором находится взаимная блокировка, приведет к повреждению списка.

Использование переменных условия

Переменная условия всегда должна использоваться вместе со взаимной блокировкой. С переменной условия может быть связана только одна взаимная блокировка, однако каждая взаимная блокировка может быть связана с несколькими переменными условия. Условие, взаимную блокировку и переменную условия можно объединить в одну структуру, как показано в следующем примере:

```
struct condition_bundle_t {
        int             condition_predicate;
        pthread_mutex_t  condition_lock;
        pthread_cond_t   condition_variable;
};
```

Ожидание выполнения условия

Взаимная блокировка, соответствующая данному условию, до начала ожидания выполнения условия должна быть деактивизирована (заблокирована). Для перевода нити в состояние ожидания события предназначены функции **pthread_cond_wait** и **pthread_cond_timedwait**. Функция автоматически

активизирует взаимную блокировку и блокирует вызывающую нить до появления сигнала о выполнения условия. После возврата запроса взаимная блокировка вновь деактивизируется.

Функция **pthread_cond_wait** блокирует нить на неопределенное время. Если условие так никогда и не будет выполнено, то нить никогда не возобновится. Поскольку в функции **pthread_cond_wait** предусмотрена точка отмены нити, единственный способ выхода из этой тупиковой ситуации - отменить блокированную нить, если такая отмена разрешена. Дополнительная информация приведена в разделе Принудительное завершение работы нити.

Функция **pthread_cond_timedwait** блокирует нить только на заданный промежуток времени. У нее есть дополнительный параметр *timeout*, который указывает дату и время снятия блокировки. Параметр *timeout* - это указатель на структуру **timespec**. Такой тип данных также называется **timestruc_t**. Он содержит следующие поля:

tv_sec
Длинное целое без знака, задающее количество секунд

tv_nsec
Длинное целое, задающее количество наносекунд

Типичный вызов функции **pthread_cond_timedwait** приведен ниже:

```
struct timespec timeout;
...
time(&timeout.tv_sec);
timeout.tv_sec += МАКС_ПРОДОЛЖИТЕЛЬНОСТЬ_ОЖИДАНИЯ;
pthread_cond_timedwait(&cond, &mutex, &timeout);
```

Параметр *timeout* задает абсолютное время снятия блокировки. Приведенный выше код показывает, каким образом можно задать не абсолютную дату, а продолжительность ожидания.

В случае применения **pthread_cond_timedwait** с указанием абсолютного времени значение поля `tv_sec` в структуре **timespec** можно вычислить с помощью функции **mktime**. В приведенном ниже примере нить будет ожидать выполнения условия до 08:00 по местному времени 1 января 2001 г.:

```
struct tm       date;
time_t          seconds;
struct timespec timeout;
...

date.tm_sec = 0;
date.tm_min = 0;
date.tm_hour = 8;
date.tm_mday = 1;
date.tm_mon = 0;         /* диапазон возможных значений: 0-11 */
date.tm_year = 101;      /* 0 означает 1900 */
date.tm_wday = 1;        /* это необязательное поле, но
                            в действительности это будет понедельник! */
date.tm_yday = 0;        /* первый день года */
date.tm_isdst = daylight;
        /* сезонное время - это внешняя переменная. Предполагается,
           что сезонное время по-прежнему будет применяться... */

seconds = mktime(&date);

timeout.tv_sec = (длинное без знака)seconds;
timeout.tv_nsec = 0L;

pthread_cond_timedwait(&cond, &mutex, &timeout);
```

Несмотря на то, что ожидание в функции **pthread_cond_timedwait** не является бесконечным, в ней предусмотрена точка отмены операции. Таким образом, ожидающую нить можно отменить, не дожидаясь тайм-аута.

Передача сигнала

Сигнал может быть передан функцией **pthread_cond_signal** или **pthread_cond_broadcast**.

Функция **pthread_cond_signal** активизирует по крайней мере одну нить, которая в данное время ожидает выполнения определенного условия. Нить для активизации выбирается в соответствии со стратегией планирования; а именно, выбирается нить с наивысшим приоритетом планирования (см. раздел Стратегия и приоритеты планирования). Учтите, что в многопроцессорных системах и некоторых операционных системах, отличных от AIX, может быть активизировано несколько нитей. Не следует рассчитывать на то, что эта функция активизирует ровно одну нить.

Функция **pthread_cond_broadcast** активизирует все нити, которые ожидают выполнения заданного условия. Однако после завершения функции нить можно вновь заблокировать до тех пор, пока не будет выполнено то же самое условие.

Если параметр *cond* задан верно, то описанные функции всегда завершают работу успешно. Однако это не означает, что какая-либо нить обязательно будет активизирована. Более того, библиотека не сохраняет информации о сигнале и о выполнении условия. Например, рассмотрим условие C. Его выполнения не ожидает ни одна нить. В момент t нить 1 сигнализирует о выполнении условия C. Вызов успешно выполняется, хотя ни одна нить не активизируется. В момент времени t+1 нить 2 вызывает функцию **pthread_cond_wait** со значением параметра *cond*, равным C. При этом нить 2 блокируется. Если ни одна из других нитей не подаст сигнала о выполнении условия C, то вполне возможно, что нить 2 будет оставаться заблокированной вплоть до завершения процесса.

Для того чтобы избежать этой тупиковой ситуации, можно проверить код ошибки **EBUSY**, который функция **pthread_cond_destroy** возвращает при удалении переменной условия. Это показано в следующем фрагменте кода:

Функция **pthread_yield** позволяет запланировать другую нить, например, одну из активизированных нитей. Дополнительная информация о функции **pthread_yield** .

Функции **pthread_cond_wait** и **pthread_cond_broadcast** нельзя применять вместе с обработчиком сигнала. Для организации ожидания сигнала в библиотеке нитей предусмотрена функция **sigwait**. Дополнительная информация о функции **sigwait**. За дополнительной информацией о функции **sigwait** обратитесь к разделу Управление сигналами.

Синхронизация нитей с помощью условных переменных

```
while (pthread_cond_destroy(&cond) == EBUSY) {
        pthread_cond_broadcast(&cond);
        pthread_yield();
}
```

Переменные условия применяются для организации ожидания выполнения определенного предиката условия. Этот предикат устанавливает другая нить, обычно та, которая сигнализирует о выполнении условия.

Семантика условия

Предикат условия должен быть защищен взаимной блокировкой. Функция ожидания (**pthread_cond_wait** или **pthread_cond_timedwait**) автоматически активизирует взаимную блокировку и блокирует нить на время ожидания. Когда поступает сигнал о выполнении условия, взаимная блокировка вновь деактивизируется и функция ожидания возвращает управление. Обратите внимание на то, что успешный возврат управления функцией вовсе не означает, что предикат истинен.

Причина этого заключается в том, что могло быть активизировано несколько нитей: либо нить вызвала функцию **pthread_cond_broadcast**, либо в результате конкуренции между двумя процессорами две нити

были активизированы одновременно. Первая нить, блокирующая взаимную блокировку, будет блокировать и все остальные активизированные нити, участвовавшие в ожидании, до тех пор, пока взаимная блокировка не будет разблокирована программой. Таким образом, к моменту, когда вторая нить получит взаимную блокировку и завершит функцию ожидания, предикат может измениться.

В общем случае каждый раз при завершении ожидания нить должна еще раз проверить предикат и определить, следует ли продолжить выполнение, возобновить ожидание или объявить тайм-аут. Помните, что само по себе завершение функции ожидания не означает, что предикат истинен (ложен).

Рекомендуется вставить функцию ожидания условия в цикл while, проверяющий предикат. В следующем фрагменте кода продемонстрирован базовый вариант такого цикла:

```
pthread_mutex_lock(&condition_lock);
while (condition_predicate == 0)
        pthread_cond_wait(&condition_variable, &condition_lock);
...
pthread_mutex_unlock(&condition_lock);
```

Семантика тайм-аута

Когда функция **pthread_cond_timedwait** завершается из-за наступления тайм-аута, предикат может быть истинным, поскольку время истечения тайм-аута может совпасть с изменением состояния предиката.

Как и в случае бесконечного ожидания, при каждом тайм-ауте нить должна проверять, не изменился ли предикат, и определять, следует ли продолжать выполнение или объявить тайм-аут. При завершении функции **pthread_cond_timedwait** рекомендуется тщательно проверять все возможные варианты. Ниже приведен пример реализации такой проверки в программе:

```
int result = CONTINUE_LOOP;

pthread_mutex_lock(&condition_lock);
while (result == CONTINUE_LOOP) {
        switch (pthread_cond_timedwait(&condition_variable,
                &condition_lock, &timeout)) {

                case 0:
                if (condition_predicate)
                        result = PROCEED;
                break;

                case ETIMEDOUT:
                result = condition_predicate ? PROCEED : TIMEOUT;
                break;

                default:
                result = ERROR;
                break;
        }
}

...
pthread_mutex_unlock(&condition_lock);
```

Выбрать действие можно с помощью переменной **result**. Рекомендуется, чтобы операторы, предшествующие освобождению взаимной блокировки, выполнялись как можно быстрее, так как взаимную блокировку нельзя удерживать в течение длительного времени.

Указав в параметре *timeout* абсолютную дату, вы можете легко установить в программе режим реального времени. В этом случае тайм-аут не нужно пересчитывать каждый раз, когда он используется в цикле (например, в том, который содержит функцию ожидания условия). В тех случаях, когда системные часы периодически переводятся оператором, запланированное ожидание (до абсолютной даты) будет закончено, как только показания системных часов превысят значение параметра *timeout*.

Пример применения переменных условия

Ниже приведен пример исходного кода процедуры точки синхронизации. *Точкой синхронизации* называется точка программы, в которой выполнение нитей приостанавливается до тех пор, пока все (или по крайней мере некоторые определенные) нити не достигнут этой точки.

Проще всего реализовать точку синхронизации с помощью счетчика, защищенного блокировкой, и переменной условия. Каждая из нитей получает блокировку, увеличивает счетчик на единицу, и если счетчик еще не достиг максимума, ожидает сигнала о выполнении условия. Если счетчик достиг максимума, то сигнал о выполнении условия передается всем нитям, и все они вновь активизируются. О выполнении условия оповещает последняя из нитей, вызывающих процедуру.

```
#define SYNC_MAX_COUNT  10

void SynchronizationPoint()
{
        /* для гарантированной инициализации применяются статические переменные */
        static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
        static cond_t  sync_cond = PTHREAD_COND_INITIALIZER;
        static int sync_count = 0;

        /* блокировка доступа к счетчику */
        pthread_mutex_lock(&sync_lock);

        /* приращение значения счетчика */
        sync_count++;

        /* проверка: следует ли продолжать ожидание */
        if (sync_count < SYNC_MAX_COUNT)

             /* ожидать остальных */
             pthread_cond_wait(&sync_cond, &sync_lock);

        else

            /* оповестить о достижении данной точки всеми нитями */
            pthread_cond_broadcast(&sync_cond);

        /* активизация взаимной блокировки - в противном случае
                из процедуры сможет выйти только одна нить! */
        pthread_mutex_unlock(&sync_lock);
}
```

У такой процедуры есть некоторые недостатки: ее можно применить только один раз и число вызывающих ее нитей обозначено символьной константой. Тем не менее, этот пример отражает основной способ применения переменных условия. Более сложные примеры применения. Более сложные примеры применения переменных условия приведены в разделе Создание сложных объектов синхронизации.

Понятия, связанные с данным:

“Стыковка с нитями” на стр. 444
Стыковка с нитью означает ожидание ее завершения. Эту операцию можно рассматривать как особый случай применения переменных условия.

Применение блокировок чтения-записи

Обычно чтение данных выполняется чаще, чем изменение и запись.

В таких случаях можно заблокировать данные таким образом, чтобы несколько нитей могли одновременно считывать данные, и только одна нить могла их изменять. Для этого предназначена блокировка типа "несколько читателей, один писатель", или блокировка чтения-записи. Блокировка чтения-записи захватывается для чтения или записи, а затем освобождается. Освободить блокировку чтения-записи может только та нить, которая ее захватила.

Объект атрибутов блокировки чтения-записи

Функция **pthread_rwlockattr_init** инициализирует объект атрибутов блокировки чтения-записи (**attr**). Значения атрибутов, применяемые по умолчанию, зависят от конкретной реализации. Вызов функции **pthread_rwlockattr_init** для уже инициализированного объекта атрибутов блокировки чтения-записи приведет к непредсказуемым результатам.

Ниже приведен пример вызова функции **pthread_rwlockattr_init** для объекта **attr**:

```
pthread_rwlockattr_t    attr;
```

и:

```
pthread_rwlockattr_init(&attr);
```

Функции, изменяющие объект атрибутов блокировки чтения-записи (в том числе функция удаления), не влияют на блокировки чтения-записи, инициализированные с помощью этого объекта.

Функция **pthread_rwlockattr_destroy** удаляет объект атрибутов блокировки чтения-записи. Применение удаленного объекта до его повторной инициализации с помощью функции **pthread_rwlockattr_init** приведет к непредсказуемым результатам. В некоторых реализациях функция **pthread_rwlockattr_destroy** может присваивать объекту, на который ссылается объект **attr**, недопустимое значение.

Создание и удаление блокировок чтения-записи

Функция **pthread_rwlock_init** присваивает блокировке чтения-записи, связанной с объектом **rwlock**, значения атрибутов из объекта **attr**. Если объекту **attr** присвоено пустое значение, то применяются атрибуты блокировки чтения-записи по умолчанию; то же самое происходит при передаче в функцию адреса объекта атрибутов блокировки чтения-записи по умолчанию. После успешной инициализации блокировка находится в состоянии "разблокирована". Инициализированная блокировка может применяться любое число раз без повторной инициализации. Вызов функции **pthread_rwlock_init** для уже инициализированной блокировки чтения-записи или применение неинициализированной блокировки приведет к непредсказуемым результатам.

Если при выполнении функции **pthread_rwlock_init** возникает ошибка, то объект **rwlock** не инициализируется. Содержимое такого объекта не определено.

Функция **pthread_rwlock_destroy** удаляет объект блокировки чтения-записи **rwlock** и освобождает заблокированные ресурсы. Ниже перечислены операции, результат выполнения которых заранее предсказать нельзя:

- Применение блокировки до ее повторной инициализации с помощью функции **pthread_rwlock_init**.
- В некоторых реализациях функция **pthread_rwlock_destroy** может присваивать объекту **rwlock** недопустимое значение. Вызов функции **pthread_rwlock_destroy** для объекта **rwlock**, который захвачен какой-либо нитью, может привести к непредвиденным результатам.
- Удаление неинициализированной блокировки чтения-записи. Удаленный объект блокировки чтения-записи можно повторно инициализировать с помощью функции **pthread_rwlock_init**. Обращение к удаленному объекту блокировки чтения-записи может привести к непредвиденным результатам.

Если вы планируете применять атрибуты блокировки чтения-записи по умолчанию, то для инициализации статической блокировки чтения-записи воспользуйтесь макрокомандой **PTHREAD_RWLOCK_INITIALIZER**. Например:

```
pthread_rwlock_t        rwlock1 = PTHREAD_RWLOCK_INITIALIZER;
```

Действие этой команды аналогично динамической инициализации с помощью функции **pthread_rwlock_init** с пустым значением параметра *attr* за исключением того, что проверка на наличие ошибок не выполняется. Например:

```
pthread_rwlock_init(&rwlock2, NULL);
```

В следующем примере продемонстрирован вызов функции **pthread_rwlock_init** с инициализированным параметром attr. Пример инициализации параметра *attr* приведен в разделе Объект атрибутов блокировки чтения-записи.

```
pthread_rwlock_init(&rwlock, &attr);
```

Захват объекта блокировки чтения-записи для чтения

Функция **pthread_rwlock_rdlock** захватывает объект блокировки чтения-записи **rwlock** для чтения. Вызывающая нить получает блокировку для чтения, если блокировка не захвачена для записи и нет нитей, ожидающих получения блокировки для записи. В случае, если блокировка не захвачена для записи, но есть нити, ожидающие получения блокировки для записи, действие функции не определено. Если блокировка захвачена для записи, то вызывающая нить не получит блокировку для чтения. Если блокировку для чтения не удалось захватить сразу, функция **pthread_rwlock_rdlock** продолжает выполняться в вызывающей нити до тех пор, пока не захватит блокировку. Если вызывающая нить уже захватила блокировку **rwlock** для записи, то результат вызова функции не определен.

Нить может несколько раз захватить блокировку **rwlock** для чтения (то есть вызвать функцию **pthread_rwlock_rdlock** *n* раз). В этом случае нить должна соответствующее число раз разблокировать объект (то есть вызвать функцию **pthread_rwlock_unlock** *n* раз).

Функция **pthread_rwlock_tryrdlock**, как и функция **pthread_rwlock_rdlock**, устанавливает блокировку для чтения. Однако в тех случаях, когда объект **rwlock** захвачен какой-либо нитью для записи, или есть нити, ожидающие получения блокировки **rwlock** для записи, эта функция возвращает сообщение об ошибке. Вызов любой из этих функций для неинициализированной блокировки чтения-записи может привести к непредвиденным результатам.

При получении сигнала нить, ожидающая захвата блокировки чтения-записи для чтения, вызывает обработчик сигнала, а после завершения его работы снова переходит в состояние ожидания блокировки.

Захват объекта блокировки чтения-записи для записи

Функция **pthread_rwlock_wrlock** захватывает объект блокировки чтения-записи **rwlock** для записи. Вызывающая нить получает блокировку для записи, если блокировка чтения-записи **rwlock** не захвачена для чтения или записи другими нитями. В противном случае функция **pthread_rwlock_wrlock** продолжает выполняться в нити до тех пор, пока не удастся захватить блокировку. Если вызывающая нить ранее уже захватила блокировку чтения-записи для чтения или записи, то попытка захватить эту же блокировку для записи может привести к непредсказуемым результатам.

Функция **pthread_rwlock_trywrlock**, как и функция **pthread_rwlock_wrlock**, захватывает блокировку для записи. Однако если объект **rwlock** уже захвачен какой-либо нитью, эта функция возвращает сообщение об ошибке. Вызов любой из этих функций для неинициализированной блокировки чтения-записи может привести к непредсказуемым результатам.

При получении сигнала нить, ожидающая захвата блокировки чтения-записи для записи, вызывает обработчик сигнала, а после завершения его работы снова переходит в состояние ожидания блокировки.

Примеры программ с применением блокировок чтения-записи

В приведенных ниже примерах программ продемонстрировано применение функций блокировки. Для запуска этих программ необходим файл **check.h** и **makefile**.

Файл **check.h**:

```
#include stdio.h
#include stdio.h
#include stdio.h
#include stdio.h
```

```
/* Простая функция, проверяющая код возврата и завершающая программу,
   если функция не была выполнена
   */
static void compResults(char *string, int rc) {
  if (rc) {
    printf("Ошибка в : %s, rc=%d",
           string, rc);
    exit(EXIT_FAILURE);
  }
  return;
}
```

Файл **Make**:

```
CC_R = xlc_r

TARGETS = test01 test02 test03

OBJS = test01.o test02.o test03.o

SRCS = $(OBJS:.o=.c)

$(TARGETS): $(OBJS)
    $(CC_R) -o $@ $@.o

clean:
    rm $(OBJS) $(TARGETS)

run:
    test01
    test02
    test03
```

Пример с одной нитью

В следующем примере функция **pthread_rwlock_tryrdlock** применяется одной нитью. Пример применения функции **Pthread_rwlock_tryrdlock** с несколькими нитями приведен в разделе Пример с несколькими нитями.

Пример: test01.c

```
#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *rdlockThread(void *arg)
{
  int rc;
  int             count=0;

  printf("Выполняется нить, получение блокировки для чтения с ожиданием\n");
  Retry:
  rc = pthread_rwlock_tryrdlock(&rwlock);
  if (rc == EBUSY) {
    if (count >= 10) {
      printf("Достигнуто ограничение на число попыток, ошибка!\n");

      exit(EXIT_FAILURE);
    }
    ++count;
    printf("Не удалось получить блокировку, выполнение других операций, а затем повтор...\n");
    sleep(1);
    goto Retry;
```

```
  }
  compResults("pthread_rwlock_tryrdlock() 1\n", rc);

  sleep(2);

  printf("освобождение блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Дополнительная нить завершена\n");
  return NULL;
}

int main(int argc, char **argv)
{
  int                   rc=0;
  pthread_t thread;

  printf("Запуск тестового набора - %s\n", argv[0]);

  printf("Главная нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);

  printf("Главная нить, создание нити для вызова функции pthread_rwlock_tryrdlock()\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread_create\n", rc);

  printf("Главная нить удерживает блокировку для записи\n");
  sleep(5);

  printf("Главная нить, освобождение блокировки для записи\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, ожидание завершения выполнения нити\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);
  printf("Главная нить завершена\n");
  return 0;
}
```

Вывод этой программы будет выглядеть приблизительно следующим образом:

```
Запуск тестового набора - ./test01
Главная нить, получение блокировки для записи
Главная нить, создание нити для вызова функции pthread_rwlock_tryrdlock()
Главная нить удерживает блокировку для записи

Выполняется нить, получение блокировки для чтения с ожиданием
Не удалось получить блокировку, выполнение других операций, а затем повтор...
Не удалось получить блокировку, выполнение других операций, а затем повтор...
Не удалось получить блокировку, выполнение других операций, а затем повтор...
Не удалось получить блокировку, выполнение других операций, а затем повтор...
Не удалось получить блокировку, выполнение других операций, а затем повтор...
Главная нить, освобождение блокировки для записи
Главная нить, ожидание завершения выполнения нити
освобождение блокировки для чтения
Дополнительная нить завершена
Главная нить завершена
```

Пример с несколькими нитями

В следующем примере функция **pthread_rwlock_tryrdlock** вызывается несколькими нитями. Пример применения функции **Pthread_rwlock_tryrdlock** в одной нити приведен в разделе Пример с одной нитью.

Пример: test01.c

```
#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *wrlockThread(void *arg)
{
  int rc;
  int             count=0;

  printf("%.8x: Выполняется нить, получение блокировки для записи\n",
         pthread_self());
  Retry:
  rc = pthread_rwlock_trywrlock(&rwlock);
  if (rc == EBUSY) {
    if (count >= 10) {
      printf("%.8x: Достигнуто ограничение на число попыток, ошибка!\n",
             pthread_self());
      exit(EXIT_FAILURE);
    }

    ++count;
    printf("%.8x: Выполнение других операций, затем повтор...\n",
           pthread_self());
    sleep(1);
    goto Retry;
  }
  compResults("pthread_rwlock_trywrlock() 1\n", rc);
  printf("%.8x: Получена блокировка для записи\n", pthread_self());

  sleep(2);

  printf("%.8x: Освобождение блокировки для записи\n",
         pthread_self());
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("%.8x: Дополнительная нить завершена\n",
         pthread_self());
  return NULL;
}

int main(int argc, char **argv)
{
  int                   rc=0;
  pthread_t             thread, thread2;

  printf("Запуск тестового набора - %s\n", argv[0]);

  printf("Главная нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);

  printf("Главная нить, создание нитей, использующих блокировки для записи\n");
  rc = pthread_create(&thread, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);

  rc = pthread_create(&thread2, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);
```

```
  printf("Главная нить удерживает блокировку для записи\n");
  sleep(1);

  printf("Главная нить, освобождение блокировки для записи\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, ожидание завершения выполнения нитей\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_join(thread2, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);
  printf("Главная нить завершена\n");
  return 0;
}
```

Вывод этой программы будет выглядеть приблизительно следующим образом:

```
Запуск тестового набора - ./test02
Главная нить, получение блокировки для записи
Главная нить, создание нитей, использующих блокировки для записи
Главная нить удерживает блокировку для записи
00000102: Выполняется нить, получение блокировки для записи
00000102: Выполнение других операций, затем повтор...
00000203: Выполняется нить, получение блокировки для записи
00000203: Выполнение других операций, затем повтор...
Главная нить, освобождение блокировки для записи
Главная нить, ожидание завершения выполнения нитей
00000102: Получена блокировка для записи
00000203: Выполнение других операций, затем повтор...
00000203: Выполнение других операций, затем повтор...
00000102: Освобождение блокировки для записи
00000102: Дополнительная нить завершена
00000203: Получена блокировка для записи
00000203: Освобождение блокировки для записи
00000203: Дополнительная нить завершена
Главная нить завершена
```

Пример захвата блокировки чтения-записи для чтения

В следующем примере продемонстрировано применение функции **pthread_rwlock_rdlock** для захвата блокировки чтения-записи для чтения:

```
Пример: test03.c

#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock;

void *rdlockThread(void *arg)
{
  int rc;

  printf("Выполняется нить, получение блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n", rc);
  printf("блокировка rwlock захвачена для чтения\n");

  sleep(5);
```

```
  printf("освобождение блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокировала\n");
  return NULL;
}

void *wrlockThread(void *arg)
{
  int rc;

  printf("Выполняется нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);

  printf("Блокировка rwlock захвачена для записи, освобождение блокировки\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокировала\n");
  return NULL;
}



int main(int argc, char **argv)
{
  int                   rc=0;
  pthread_t             thread, thread1;

  printf("Запуск тестового набора - %s\n", argv[0]);

  printf("Главная нить, инициализация блокировки чтения-записи\n");
  rc = pthread_rwlock_init(&rwlock, NULL);
  compResults("pthread_rwlock_init()\n", rc);

  printf("Главная нить, захват блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n",rc);

  printf("Главная нить, повторный захват этой же блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock() second\n", rc);

  printf("Главная нить, создание нити для захвата блокировки для чтения\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread_create\n", rc);

  printf("Главная нить - освобождение первой блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, создание нити для захвата блокировки для записи\n");
  rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);

  sleep(5);
  printf("Главная нить - освобождение второй блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, ожидание завершения нитей\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_join(thread1, NULL);
  compResults("pthread_join\n", rc);
```

```
  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);

  printf("Главная нить завершена\n");
  return 0;
}
```

Вывод этой программы будет выглядеть приблизительно следующим образом:

```
$ ./test03
Запуск тестового набора - ./test03
Главная нить, инициализация блокировки чтения-записи
Главная нить, захват блокировки для чтения
Главная нить, повторный захват этой же блокировки для чтения
Главная нить, создание нити для захвата блокировки для чтения
Главная нить - освобождение первой блокировки для чтения
Главная нить, создание нити для захвата блокировки для записи
Выполняется нить, получение блокировки для чтения
блокировка rwlock захвачена для чтения
Выполняется нить, получение блокировки для записи
Главная нить - освобождение второй блокировки для чтения
Главная нить, ожидание завершения нитей
освобождение блокировки для чтения
Дополнительная нить разблокировала
Блокировка rwlock захвачена для записи, освобождение блокировки
Дополнительная нить разблокировала
Главная нить завершена
```

Стыковка с нитями

Стыковка с нитью означает ожидание ее завершения. Эту операцию можно рассматривать как особый случай применения переменных условия.

Ожидание завершения нити

Функция **pthread_join** позволяет нити дождаться завершения другой нити. В более сложной ситуации, когда требуется дождаться завершения нескольких нитей, можно воспользоваться переменными условия.

Вызов функции pthread_join

Функция **pthread_join** блокирует вызывающую нить до завершения указанной нити. При этом целевая (т.е. вызываемая) нить не должна быть автономной (помеченной на освобождение ресурсов). Если целевая нить уже завершилась, но не является автономной, то возврат из процедуры **pthread_join** происходит немедленно. После стыковки с целевой нитью она автоматически становится автономной, и отведенную ей память можно очистить.

В приведенной ниже таблице описаны возможные случае вызова функции **pthread_join** из нити, в зависимости от значения атрибутов **state** и **detachstate** целевой нити.

Целевое состояние	Целевая нить не автономна	Целевая нить автономна
Целевая нить активна	Вызывающая нить блокирована до завершения целевой нити.	Возврат из функции происходит немедленно с выдачей сообщения об ошибке.
Целевая нить завершена	Возврат из функции происходит немедленно с выдачей сообщения об успешном завершении.	

Стыковка нескольких нитей с одной

Несколько нитей могут стыковаться с одной и той же нитью, если она не автономна. Успех этой операции зависит от порядка вызовов процедуры **pthread_join** и от того, в какой момент целевая нить завершится.

- Любой вызов процедуры **pthread_join** до завершения целевой нити блокирует вызывающую нить.
- После завершения целевой нити все заблокированные нити освобождаются, а целевая нить автоматически становится автономной.
- Любой вызов процедуры **pthread_join** после завершения целевой нити приводит к ошибке, так как нить уже стала автономной в результате предыдущей стыковки.
- Если до завершения целевой нити процедура **pthread_join** не вызывалась, то при первом вызове этой процедуры будет немедленно сообщено об успешном завершении, а все последующие вызовы будут приводить к ошибке.

Пример стыковки

В этом примере выполнение программы завершается после того, как будет выдано по пять сообщений на каждом языке. Это достигается за счет блокировки главной нити до завершения нити "writer".

```
#include <pthread.h>    /* первый включаемый файл - pthread.h */
#include <stdio.h>      /* поддержка функции printf() */

void *Thread(void *string)
{
        int i;

        /* записывает пять сообщений и завершается */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_attr_t attr;
        pthread_t e_th;
        pthread_t f_th;

        int rc;

        /* создает правильный атрибут */
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_UNDETACHED);

        /* создание обеих нитей */
        rc = pthread_create(&e_th, &attr, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, &attr, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_attr_destroy(&attr);

        /* стыкует нити */
        pthread_join(e_th, NULL);
        pthread_join(f_th, NULL);

        pthread_exit(NULL);
}
```

Нить не может стыковаться сама с собой, так как это приведет к возникновению тупиковой ситуации. Подобные ошибки обнаруживаются библиотекой. Однако две нити могут стыковаться друг с другом. Это также приведет к возникновению тупиковой ситуации, но такие ошибки не обнаруживаются библиотекой.

Возврат информации из нити

Процедура **pthread_join** позволяет также передавать информацию из одной нити в другую. При вызове функции **pthread_exit** или при завершении работы процедуры входа нить возвращает указатель (см. раздел Нормальное завершение работы нити). Этот указатель хранится до тех пор, пока нить не станет автономной, и процедура **pthread_join** может вернуть его.

Например, команда **grep** с несколькими нитями может быть реализована так, как показано в следующем примере. В этом примере главная нить создает по одной нити для каждого просматриваемого файла, причем во всех нитях применяется одинаковая процедура точки входа. После этого главная нить ожидает завершения всех остальных нитей. Каждая дочерняя нить заносит найденные строки в динамический буфер (свой для каждой нити) и возвращает указатель на этот буфер. Главная нить печатает содержимое всех буферов и освобождает их.

```
/* дочерняя нить для просмотра */
...
buffer = malloc(...);
        /* ищет в файле строки, совпадающие с шаблоном
           поиска, и заносит их в буфер             */
return (buffer);

/* главная нить   */
...
for (/* для каждой созданной нити */) {
        void *buf;
        pthread_join(thread, &buf);
        if (buf != NULL) {
                /* печатает все строки из буфера, указывая
                        перед каждой строкой имя файла нити */
                free(buf);
        }
}
...
```

При принудительном завершении целевой нити процедура **pthread_join** возвращает в указателе -1 (см. раздел Отмена нити). Поскольку указатель не может быть равен -1, это означает, что нить была завершена принудительно.

Возвращенный указатель может ссылаться на данные любого типа. После завершения нити и освобождения ее памяти значение указателя должно оставаться действительным. Следовательно, не следует возвращать значение, так как при освобождении памяти нити вызывается деструктор.

При возврате указателя на область динамической памяти, отведенную для нескольких нитей, следует учесть некоторые особенности. Рассмотрим следующий фрагмент кода:

```
void *returned_data;
...
pthread_join(target_thread, &returned_data);
/* считывание информации из области returned_data */
free(returned_data);
```

Указатель **returned_data** будет освобожден, если этот фрагмент кода выполняется только одной нитью. Если же этот фрагмент кода одновременно выполняется несколькими нитями, указатель **returned_data** будет освобожден несколько раз, что недопустимо. Для предотвращения такой ситуации предусмотрите флаг, защищенный взаимной блокировкой, который будет сигнализировать об освобождении указателя **returned_data**. Следующую строку из предыдущего примера:

```
free(returned_data);
```

следует заменить на следующий фрагмент кода, в котором для блокирования доступа к критической области применяется взаимная блокировка (предполагается, что начальное значение переменной *flag* равно 0):

```
/* установка блокировки - критический фрагмент кода, другие нити
                не должны параллельно выполнять этот код */
if (!flag) {
        free(returned_data);
        flag = 1;
}
/* снятие блокировки - критический фрагмент окончен */
```

Блокирование доступа к критической области гарантирует, что указатель **returned_data** будет освобожден только один раз.

При возврате указателя на область динамической памяти, отведенную для нескольких нитей, которые выполняют различный код, следует освободить указатель в одной и только в одной нити.

Понятия, связанные с данным:

“Использование переменных условия” на стр. 430
Переменные условия позволяют приостановить выполнение нити, пока не наступит заданное событие или не будет соблюдено некоторое условие.

“Информация о нити” на стр. 457
Во многих приложениях необходимо работать с данными, относящимися к отдельным нитям.

Планирование нитей

В библиотеке нитей предусмотрен набор функций для планирования нитей и управления планированием.

Кроме того, имеются функции для управления планированием нитей во время операций синхронизации, например, при установке взаимной блокировки. У каждой нити есть свой набор параметров планирования. Эти параметры можно установить с помощью объекта атрибутов нити еще до создания самой нити. Параметры можно динамически изменять во время обработки нити.

Управление планированием нитей часто представляет собой сложную задачу. Поскольку планировщик обрабатывает все нити в системе, параметры планирования конкретной нити взаимодействуют с параметрами планирования других нитей в этом и других процессах. Если вы хотите управлять планированием нитей, в первую очередь используйте перечисленные ниже средства.

С помощью библиотеки нитей программист может управлять планированием обработки нитей следующими способами:

- Настройка атрибутов планирования при создании нити
- Динамическое изменение атрибутов планирования уже созданной нити
- Определение влияния взаимной блокировки на планирование нитей при создании взаимной блокировки (*планирование синхронизации*).
- Динамическое изменение атрибутов планирования нити в ходе синхронизации (*планирование синхронизации*).

Параметры планирования

У нити есть следующие параметры планирования:

Параметр	Описание
область действия	Область действия нити определяется моделью обработки нитей, используемой библиотекой.
стратегия	Стратегия планирования нити определяет, как планировщик работает с нитью после передачи ей управления.
приоритет	Приоритет планирования нити определяет относительную степень важности ее выполнения.

Параметры планирования можно устанавливать как до создания нити, так и в ходе ее выполнения. Как правило, необходимо управлять параметрами планирования только для нитей, требующих много ресурсов процессора. Поэтому значения по умолчанию, устанавливаемые библиотекой нитей, приходится изменять достаточно редко.

Работа с атрибутом inheritsched

Атрибут **inheritsched** объекта атрибутов нити указывает способ определения атрибутов планирования нити. Возможны следующие значения:

Значения	Описание
PTHREAD_INHERIT_SCHED	Означает, что новая нить наследует набор атрибутов планирования (атрибуты **schedpolicy** и **schedparam**) от породившей ее нити. Атрибуты планирования, указанные в объекте атрибутов, игнорируются.
PTHREAD_EXPLICIT_SCHED	Означает, что атрибуты планирования вновь созданной нити берутся из соответствующего объекта атрибутов.

Значение атрибута **inheritsched** по умолчанию равно PTHREAD_INHERIT_SCHED. Значение атрибута изменяется с помощью функции **pthread_attr_setinheritsched**. Функция **pthread_attr_getinheritsched** возвращает текущее значение атрибута.

Для установки значений атрибутов планирования в объекте атрибутов нити следует сначала установить значение **inheritsched** равным PTHREAD_EXPLICIT_SCHED. В противном случае значения атрибутов планирования из объекта атрибутов будут проигнорированы.

Стратегия и приоритеты планирования

В библиотеке нитей реализованы следующие стратегии планирования:

Библиотека	Описание
SCHED_FIFO	Планирование по принципу простой очереди "первый вошел, первый вышел" (FIFO). Каждой нити присваивается постоянный приоритет; если несколько нитей имеют один и тот же приоритет, они выполняются в порядке FIFO.
SCHED_RR	Планирование по принципу "карусели" (RR). Каждой нити присваивается постоянный приоритет; если несколько нитей имеют один и тот же приоритет, им в порядке FIFO выделяются равные значения времени процессора.
SCHED_OTHER	Способ планирования, принятый в AIX по умолчанию. Каждой нити присваивается приоритет, который планировщик может динамически изменять в соответствии с уровнем ее активности; каждой нити выделяется определенное число квантов времени. В других системах могут применяться другие стратегии планирования.

До AIX версии 5.3 изменение приоритета нити при указании стратегии планирования **SCHED_OTHER** недопустимо. В этом случае ядро напрямую управляет приоритетом и функции **pthread_setschedparam** можно передать только значение **DEFAULT_PRIO**. Значение **DEFAULT_PRIO** определено в файле **pthread.h** в качестве 1. Прочие значения игнорируются.

Начиная с AIX 5.3, вы можете изменить приоритет нити при указании стратегии планирования **SCHED_OTHER**. Функции **pthread_setschedparam** можно передавать значение от 40 до 80, однако значения, превышающие 60, могут указывать только привилегированные пользователи. Приоритет в диапазоне от 1 до 39 соответствует приоритету 40, а приоритет от 81 до 127 - приоритету 80.

Примечание: Ядро AIX инвертирует значения уровня приоритета. В ядре AIX возможные значения приоритета - от 0 до 127, причем 0 означает наивысший приоритет, а 127 - низший. Команда **ps** и подобные ей возвращают значения приоритета ядра.

В библиотеке нитей для управления приоритетами предусмотрена структура **sched_param**, определенная в файле заголовка **sys/sched.h**. Эта структура содержит следующие поля:

Поля	Описание
sched_priority	Задает приоритет.
sched_policy	Это поле игнорируется библиотекой нитей. Не используйте его.

Установка стратегии планирования и приоритета при создании нити

Для задания стратегии планирования при создании нити следует установить соответствующее значение атрибута **schedpolicy** в объекте атрибутов нити. С помощью функции **pthread_attr_setschedpolicy** можно присвоить атрибуту schedpolicy любое из определенных ранее значений. Текущее значение атрибута **schedpolicy** для данной нити можно узнать с помощью функции **pthread_attr_getschedpolicy**.

Для задания приоритета планирования при создании нити следует установить соответствующее значение атрибута **schedparam** в объекте атрибутов нити. Функция **pthread_attr_setschedparam** устанавливает значение атрибута **schedparam**, копируя его из указанной структуры. Функция **pthread_attr_getschedparam** возвращает текущее значение атрибута **schedparam**.

В приведенном ниже фрагменте кода описывается создание нити со стратегией планирования "карусель" и уровнем приоритета 3:

```
sched_param schedparam;

schedparam.sched_priority = 3;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_RR);
pthread_attr_setschedparam(&attr, &schedparam);

pthread_create(&thread, &attr, &start_routine, &args);
pthread_attr_destroy(&attr);
```

Атрибут **inheritsched** подробно описан в разделе Работа с атрибутом inheritsched.

Установка атрибутов планирования во время выполнения

Функция **pthread_getschedparam** возвращает значения атрибутов нити **schedpolicy** и **schedparam**. Установить эти атрибуты можно с помощью функции **pthread_setschedparam**. При изменении стратегии и приоритета нити, которая в этот момент обрабатывается процессором, новые значения вступят в силу при следующей обработке нити. Если нить в данный момент не обрабатывается, она может быть запущена сразу после вызова функции установки атрибутов.

Допустим, например, для нити T применяется стратегия планирования round-robin, и во время обработки этой нити значение ее атрибута **schedpolicy** изменяется на FIFO. Нить T будет выполняться до конца выделенного ей времени, после чего вступят в силу новые атрибуты планирования. Если нити с более высоким приоритетом отсутствуют, T будет запущена еще раз, даже есть другие нити с тем же приоритетом, что и T. Другой пример: пусть нить имеет низкий приоритет и в данный момент не обрабатывается. Если другая нить повысит ее приоритет с помощью функции **pthread_setschedparam**, то при отсутствии нитей с более высоким приоритетом исходная нить будет запущена сразу же.

Примечание: В обеих рассмотренных функциях применяются два параметра: параметр *policy* и структура **sched_param**. Хотя в этой структуре и есть поле `sched_policy`, программы не должны работать с ним. Рассмотренные функции передают информацию о стратегии планирования в параметре *policy*, игнорируя поле `sched_policy`.

Сведения о стратегии планирования

В приложениях следует применять стратегию планирования по умолчанию, кроме специальных приложений, для которых необходимо применять постоянные приоритеты. При работе со стратегиями, отличными от стратегии по умолчанию, учтите следующее:

- При применении стратегии round-robin все нити с одинаковым приоритетом, независимо от уровня их активности, находятся в равном положении. Это может оказаться полезным при написании программ, в которых необходимо опрашивать датчики или периодически записывать информацию на внешние устройства.
- Стратегию FIFO следует применять с большой осторожностью. Нить, для которой применяется стратегия FIFO, будет выполняться без перерывов до самого завершения, если она не блокирована вызовом такой операции, как ввод или вывод. Прервать выполнение нити со стратегией FIFO и высоким приоритетом нельзя, и такая нить может значительно снизить общую производительность системы. Никогда не задавайте стратегию FIFO для нитей с большим объемом вычислений (например, обращение большой матрицы) и им подобных.

Стратегия планирования и приоритет нити зависят также и от ее области действия. Применять стратегии FIFO и round-robin можно не всегда.

sched_yield, функция

Функция **sched_yield** - это эквивалент функции **yield** для нитей. Функция **sched_yield** принудительно освобождает процессор и предоставляет возможность планирования других нитей. Следующая запланированная нить может относиться к тому же или к другому процессу. Функцию **yield** не следует использовать в программах с несколькими нитями.

Интерфейс функции **pthread_yield** недоступен в спецификации Single UNIX Specification версии 2.

Понятия, связанные с данным:

“Планирование синхронизации” на стр. 452
Управление приоритетом нитей требуется в том случае, если существуют ограничения, особенно ограничения по времени, требующие, чтобы некоторые нити выполнялись быстрее, чем остальные.

“Список функций планирования”
В этом разделе перечислены функции планирования.

“Создание программ с несколькими нитями” на стр. 485
Создание программ с несколькими нитями аналогично разработке программ, состоящих из нескольких процессов. Процесс разработки программы включает в себя компиляцию и отладку исходного кода.

Список функций планирования

В этом разделе перечислены функции планирования.

Функция	Описание
pthread_attr_getschedparam	Возвращает значение атрибута **schedparam** из объекта атрибутов нити.
pthread_attr_setschedparam	Устанавливает значение атрибута **schedparam** в объекте атрибутов нити.
pthread_getschedparam	Возвращает значения атрибутов нити **schedpolicy** и **schedparam**.
sched_yield	Освобождает процессор вызвавшей нити.

Понятия, связанные с данным:

“Планирование нитей” на стр. 447
В библиотеке нитей предусмотрен набор функций для планирования нитей и управления планированием.

Область действия и уровень параллелизма

Область действия определяет способ установки соответствия между пользовательской нитью и нитью ядра.

. В библиотеке нитей определены следующие возможные области действия:

PTHREAD_SCOPE_PROCESS
Процесс (или *локальная область действия*). Это означает, что при запуске нити учитываются все остальные нити процесса с локальной областью действия. Пользовательская нить уровня процесса - это пользовательская нить, которая работает с нитью ядра совместно с другими пользовательскими нитями того же процесса (также уровня процесса). Все пользовательские нити в модели M:1 - это нити уровня процесса.

PTHREAD_SCOPE_SYSTEM
Системная (или *глобальная*) область действия. Это означает, что при запуске нити учитываются все остальные нити в системе, и нить напрямую присваивается одной из нитей ядра. Все пользовательские нити в модели 1:1 - это нити системного уровня.

В модели обработки нитей M:N пользовательские нити могут соответствовать как процессу, так и системе. Поэтому модель обработки нитей M:N часто называют *смешанной моделью*.

Уровень параллелизма - это свойство библиотек нитей в модели M:N. Уровень определяет число виртуальных процессоров, применяемых для обработки пользовательских нитей в области процесса. Это число не должно превышать число пользовательских нитей в области процесса, обычно оно динамически изменяется библиотекой нитей. Кроме того, количество доступных нитей ядра в системе также ограничено.

Задание области действия

Область действия нити можно задать только до ее создания. Для этого следует установить соответствующее значение атрибута contention-scope в объекте атрибутов нити. Функция **pthread_attr_setscope** устанавливает значение этого атрибута, а функция **pthread_attr_getscope** возвращает его текущее значение.

Область действия нитей имеет значение только в реализации библиотеки для смешанной модели обработки нитей M:N. Процедура **TestImplementation** может выглядеть так:

```
int TestImplementation()
{
        pthread_attr_t a;
        int result;

        pthread_attr_init(&a);
        switch (pthread_attr_setscope(&a, PTHREAD_SCOPE_PROCESS))
        {
                case 0:          result = LIB_MN; break;
                case ENOTSUP:    result = LIB_11; break;
                case ENOSYS:     result = NO_PRIO_OPTION; break;
                default: result = ERROR; break;
        }
```

```
        pthread_attr_destroy(&a);
        return result;
}
```

Влияние области действия на планирование

Область действия нити влияет на параметры ее планирования. Каждая нить определенного уровня связана с отдельной нитью ядра. Поэтому при изменении стратегии планирования и приоритета глобальной пользовательской нити изменяются стратегия планирования и уровень приоритета соответствующей нити ядра.

В AIX стратегии планирования с постоянным приоритетом (FIFO и round-robin) могут применяться только для нитей ядра с правами root. Следующий код возвратит код ошибки **EPERM** (при условии, что вызывающая нить относится к системной области, но не имеет прав доступа root). Если же область действия вызывающей нити - процесс, то ошибка не возникнет.

```
schedparam.sched_priority = 3;
pthread_setschedparam(pthread_self(), SCHED_FIFO, schedparam);
```

Примечание: Для управления параметрами планирования пользовательских нитей уровня процесса права доступа root не требуются.

Для пользовательских нитей уровня процесса можно устанавливать любые допустимые стратегии и приоритеты. Однако способы планирования двух нитей с одинаковыми стратегиями и приоритетами, но с различными областями действия, будут различными. Нити уровня процесса выполняются нитями ядра, параметры планирования которых устанавливает библиотека.

Понятия, связанные с данным:

“Сведения о нитях и процессах” на стр. 407
Нить - это независимый поток управления, работающий в том же адресном пространстве, что и остальные независимые потоки управления в рамках процесса.

Планирование синхронизации

Управление приоритетом нитей требуется в том случае, если существуют ограничения, особенно ограничения по времени, требующие, чтобы некоторые нити выполнялись быстрее, чем остальные.

С помощью объектов синхронизации, например, взаимных блокировок, можно приостановить выполнение даже нитей с высоким приоритетом. При этом в некоторых случаях может возникнуть нежелательный эффект, который называется *инверсией приоритетов*. Для его предотвращения в библиотеке нитей предусмотрено специальное средство, *протоколы взаимных блокировок*.

Планирование при синхронизации определяет изменение параметров планирования, в частности, приоритета, при взаимной блокировке. Это позволяет программисту управлять параметрами планирования и предотвратить изменение приоритетов, что весьма полезно при работе со сложными схемами блокировки. В некоторых реализациях библиотеки нитей планирование при синхронизации отсутствует.

Инверсия приоритетов

Инверсия приоритетов возникает тогда, когда нить с низким приоритетом захватывает взаимную блокировку, приостанавливая выполнение нити с высоким приоритетом. Такая нить может неограниченное время сохранять взаимную блокировку, так как ее приоритет низкий. Из-за этого не удается обеспечить выполнение нити за заданный срок.

В приведенном ниже примере показан типичный случай инверсии приоритетов. В нем рассмотрен случай однопроцессорной системы. В многопроцессорных системах инверсия приоритетов возникает в аналогичных ситуациях.

В этом примере взаимная блокировка *M* защищает некоторый общий ресурс. Приоритет нити *A* равен 100, поэтому она часто добавляется планировщиком в очередь на выполнение. Приоритет нити *B* равен 20. Эта нить выполняется в фоновом режиме. Приоритет остальных нитей процесса - около 60. Ниже приведен фрагмент кода нити *A*:

```
pthread_mutex_lock(&M);             /* 1 */
...
pthread_mutex_unlock(&M);
```

Ниже приведен фрагмент кода нити *B*:

```
pthread_mutex_lock(&M);          /* 2 */
...
fprintf(...);                    /* 3 */
...
pthread_mutex_unlock(&M);
```

Рассмотрим следующий порядок выполнения нитей: запускается нить *B* и выполняется строка 2. При выполнении строки 3 нить *B* останавливается и запускается нить *A*. В ней выполняется только первая строка, после чего нить блокируется, так как взаимная блокировка *M* захвачена нитью *B*. В результате будут выполняться другие нити процесса. Поскольку нити *B* назначен очень низкий приоритет, ее выполнение может возобновиться только через длительное время, в течение которого, несмотря на свой высокий приоритет, будет блокирована нить *A*.

Протоколы взаимных блокировок

Для предотвращения инверсии приоритетов в библиотеке нитей предусмотрены следующие протоколы взаимных блокировок:

Протокол наследования приоритета
Иногда этот протокол называется *базовым протоколом наследования приоритета*. В соответствии с этим протоколом владелец взаимной блокировки наследует приоритет, который является максимальным среди приоритетов заблокированных нитей. Если нить блокируется при попытке захватить взаимную блокировку, владелец этой блокировки временно получает приоритет блокированной нити, если он больше его собственного приоритета. При освобождении взаимной блокировки приоритет владельца восстанавливается.

Протокол защиты приоритета
Иногда этот протокол называется *эмуляцией протокола максимального приоритета*. В протоколе защиты приоритетов с каждой взаимной блокировкой связан *максимальный приоритет*. Это некоторое допустимое значение приоритета. Когда нить захватывает взаимную блокировку, ей временно присваивается максимальный приоритет этой блокировки, если он превышает ее собственный приоритет. При освобождении взаимной блокировки приоритет владельца восстанавливается. Максимальный приоритет должен равняться наибольшему значению среди приоритетов нитей, которые могут захватить взаимную блокировку. Если его значение будет меньше, то протокол не будет защищать от инверсии приоритетов и тупиков.

Оба протокола увеличивают приоритет нити, захватившей взаимную блокировку, что позволяет гарантировать выполнение нити за указанный срок. Кроме того, протоколы взаимных блокировок предотвращают появление тупиков. С каждой взаимной блокировкой связывается свой протокол взаимных блокировок.

Выбор протокола взаимных блокировок

Протокол взаимных блокировок настраивается при создании взаимной блокировки путем задания атрибутов. Протокол взаимных блокировок задается с помощью специального атрибута протокола. Этот атрибут можно настроить в объекте атрибутов взаимной блокировки с помощью функций **pthread_mutexattr_getprotocol** и **pthread_mutexattr_setprotocol**. Допустимы следующие значения атрибута протокола:

Значение	Описание
PTHREAD_PRIO_DEFAULT	Нет значения
PTHREAD_PRIO_NONE	Протокол не выбран.
PTHREAD_PRIO_INHERIT	Протокол наследования приоритетов.
PTHREAD_PRIO_PROTECT	Протокол защиты приоритетов.

Примечание: Поведение **PTHREAD_PRIO_DEFAULT** такое же, как у атрибута **PTHREAD_PRIO_INHERIT**. С помощью ссылки на взаимную блокировку, нити работают с атрибутом по умолчанию, который временно повышает приоритет владельца блокировки, когда пользователь заблокирован и имеет более высокий приоритет, чем владелец. Поэтому, возможно только три типа поведения, хотя существует четыре значения, возможных для приоритета в структуре атрибута.

В протоколе защиты приоритетов требуется настроить дополнительный атрибут. Он задает максимальный приоритет взаимной блокировки. Этот атрибут можно настроить в объекте атрибутов взаимной блокировки с помощью функций **pthread_mutexattr_getprioceiling** и **pthread_mutexattr_setprioceiling**.

Для динамического изменения максимального приоритета взаимной блокировки служат функции **pthread_mutex_getprioceiling** и **pthread_mutex_setprioceiling**. При динамическом изменении приоритета взаимная блокировка захватывается библиотекой. Для предотвращения тупика блокировка не должна принадлежать нити, вызывающей функцию **pthread_mutex_setprioceiling**. Динамическое изменение максимального приоритета взаимной блокировки требуется при увеличении приоритета нити.

Реализации протоколов взаимных блокировок относятся к дополнительным функциям системы. Эти протоколы являются компонентами POSIX.

Выбор протокола наследования или защиты

Оба протокола выполняют одинаковые функции и основаны на увеличении приоритета нити, захватывающей взаимную блокировку. При наличии обоих протоколов нужно выбрать один из них. Выбор протокола зависит от того, известны ли приоритеты нитей, захватывающих взаимную блокировку, тому программисту, который эту блокировку создает. Обычно взаимные блокировки, которые определены в библиотеке и применяются прикладными нитями, работают с протоколом наследования, а взаимные блокировки, созданные в прикладных программах, работают с протоколом защиты.

На выбор могут повлиять и повышенные требования к производительности программы. В большинстве реализаций, особенно в AIX, для изменения приоритета нити необходимо выполнить системный вызов. Протоколы взаимных блокировок различаются по числу генерируемых системных вызовов:

- В протоколе наследования системный вызов выполняется каждый раз, когда нить блокируется при попытке захватить взаимную блокировку.
- В протоколе защиты системный вызов выполняется при каждом захвате взаимной блокировки нитью.

В общем случае для повышения производительности программы рекомендуется выбрать протокол наследования, так как нити конкурируют за взаимную блокировку достаточно редко. Взаимные блокировки захватываются на небольшие периоды времени, поэтому вероятность блокировки нити при попытке захвата взаимной блокировки достаточно низкая.

Понятия, связанные с данным:

“Планирование нитей” на стр. 447
В библиотеке нитей предусмотрен набор функций для планирования нитей и управления планированием.

Список функций синхронизации

В этом разделе перечислены функции синхронизации.

`pthread_mutex_destroy`
Удаляет взаимную блокировку.

pthread_mutex_init
Инициализирует взаимную блокировку и задает ее атрибуты.

PTHREAD_MUTEX_INITIALIZER
Инициализирует статическую взаимную блокировку с атрибутами по умолчанию.

pthread_mutex_lock и pthread_mutex_trylock
Захватывает взаимную блокировку.

pthread_mutex_unlock
Освобождает взаимную блокировку.

pthread_mutexattr_destroy
Удаляет объект атрибутов взаимной блокировки.

pthread_mutexattr_init
Создает объект атрибутов взаимной блокировки и инициализирует его значениями по умолчанию.

pthread_cond_destroy
Удаляет переменную условия.

pthread_cond_init
Инициализирует переменную условия и задает ее атрибуты.

PTHREAD_COND_INITIALIZER
Инициализирует статическую переменную условия с атрибутами по умолчанию.

pthread_cond_signal или pthread_cond_broadcast
Разблокирует одну или несколько нитей, заблокированных с помощью переменной условия.

pthread_cond_wait и pthread_cond_timedwait
Блокирует вызывающую нить при выполнении условия.

pthread_condattr_destroy
Удаляет объект атрибутов условия.

pthread_condattr_init
Создает объект атрибутов условия и инициализирует его значениями по умолчанию.

Разовая инициализация

Некоторые библиотеки на языке C используют динамическую инициализацию, когда глобальная инициализация библиотеки выполняется при первом вызове процедуры из этой библиотеки.

В программах с одной нитью это реализуется с помощью статической переменной, значение которой проверяется при запуске каждой процедуры, как показано в приведенном ниже фрагменте кода:

```
static int isInitialized = 0;
extern void Initialize();

int function()
{
        if (isInitialized == 0) {
                Initialize();
                isInitialized = 1;
        }
        ...
}
```

Для динамической инициализации библиотеки в программах с несколькими нитями недостаточно простого флага инициализации. Этот флаг должен быть защищен от изменения при одновременном вызове функции библиотеки несколькими нитями. Для защиты флага требуется применять механизм взаимных блокировок (семафоров), однако и эти семафоры необходимо предварительно инициализировать. Таким образом, требование разовой инициализации семафоров возвращает к исходной проблеме.

Для того чтобы сохранить прежнюю структуру в программах с несколькими нитями, воспользуйтесь функцией **pthread_once**. Если такой вариант вас не устраивает, вы можете выполнять инициализацию явно, вызывая перед каждым применением библиотеки экспортированную функцию инициализации. Функцию **pthread_once** можно применять вместо взаимных блокировок и условных переменных.

Объект разовой инициализации

Однократность инициализации гарантируется специальным объектом инициализации. Этот объект представляет собой переменную типа **pthread_once_t**. В AIX и большинстве других реализаций библиотеки нитей тип данных **pthread_once_t** является структурой.

Объект разовой инициализации обычно является глобальной переменной. Эта переменная инициализируется макрокомандой **PTHREAD_ONCE_INIT**, как показано в следующем примере:

```
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

Инициализация может быть выполнена как в начальной, так и в любой другой нити. В одной программе может применяться несколько объектов однократной инициализации. К ним предъявляется только одно требование - они должны быть инициализированы макрокомандой.

Процедура разовой инициализации

При первом вызове функция **pthread_once** вызывает процедуру инициализации, связанную с указанным объектом разовой инициализации. При всех последующих вызовах она не выполняет никаких действий. С каждым объектом разовой инициализации должна быть связана только одна процедура инициализации. Прототип процедуры инициализации должен выглядеть следующим образом:

```
void init_routine();
```

Функция **pthread_once** не содержит точки отмены операции. Однако процедура инициализации может включать точки отмены. Если отмена разрешена, то первую нить, вызвавшую функцию **pthread_once**, можно прервать во время выполнения процедуры инициализации. В этом случае процедура не будет считаться выполненной, и при следующем вызове функции **pthread_once** процедура инициализации будет вызвана повторно.

При создании функций разовой инициализации, особенно выполняющих неидемпотентные операции, такие как открытие файла, установка взаимной блокировки или выделение памяти, рекомендуется регистрировать функции очистки.

Процедуры разовой инициализации могут применяться для инициализации взаимных блокировок или условных переменных, а также для выполнения динамической инициализации. В библиотеке с поддержкой нескольких нитей приведенный выше фрагмент кода (`void init_routine();`) будет выглядеть следующим образом:

```
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
extern void Initialize();

int function()
{
        pthread_once(&once_block, Initialize);
        ...
}
```

Понятия, связанные с данным:

“Завершение работы нитей” на стр. 416
Нить автоматически завершает работу при возврате из процедуры точки входа.

“Информация о нити” на стр. 457
Во многих приложениях необходимо работать с данными, относящимися к отдельным нитям.

“Написание реентерабельных программ и программ с поддержкой нитей” на стр. 479
В процессах с одной нитью только один поток управления, и обеспечивать реентерабельность кода или поддержку нитей не требуется. В процессах с несколькими нитями одни и те же функции и ресурсы могут использоваться несколькими потоками управления одновременно.

Информация о нити

Во многих приложениях необходимо работать с данными, относящимися к отдельным нитям.

Например, для выполнения команды **grep**, если на каждый файл отводится одна нить, необходимы обработчики файлов для конкретных нитей и список найденных строк. В этих целях библиотека нитей содержит интерфейс данных для конкретных нитей.

Данные для конкретных нитей можно просмотреть в виде двумерного массива значений, причем по строкам размещены ключи, а по столбцам - ИД нитей. *Ключ* данных конкретной нити - это объект со скрытой реализацией типа **pthread_key_t**. Один и тот же ключ может применяться во всех нитях процесса. Хотя ключи для всех нитей один и те же, с этими ключами в различных нитях связаны различные данные. Благодаря этому они могут указывать на любой тип данных, например, на динамические строки или структуры.

На следующем рисунке в нити T2 значение данных 12 связано с ключом K3. В нити T4 с тем же ключом связано значение 2.

Ключи	**Нить T1**	**Нить T2**	**Нить T3**	**Нить T4**
K1	6	56	4	1
K2	87	21	0	9
K3	23	**12**	61	**2**
K4	11	76	47	88

Создание и уничтожение ключей

Ключи данных для конкретных нитей необходимо создать до первого обращения. После завершения работы нити значения ее ключей автоматически уничтожаются. Кроме того, ключ можно уничтожить и по запросу на освобождение отведенной под него памяти.

Создание ключа

Для создания ключа, связанного с данными нити, нужно вызвать функцию **pthread_key_create**. Эта функция возвращает ключ. Значения данных, соответствующих ключу, при этом равны **NULL** для всех нитей, в том числе и для тех нитей, которые еще не созданы.

Пусть, например, есть две нити (*A* и *B*). Нить *A* выполняет в хронологическом порядке следующие операции:

1. Создает ключ данных *K*.

   Нити *A* и *B* могут использовать ключ *K*. В обеих нитях этому ключу соответствует значение **NULL**.
2. Создает нить *C*.

   Нить *C* также может использовать ключ *K*. В нити *C* этому ключу соответствует значение **NULL**.

Максимальное число ключей для одного процесса равно 450. Это значение можно получить с помощью символьной константы **PTHREAD_KEYS_MAX**.

Функцию **pthread_key_create** можно вызвать только один раз, иначе будет создано два разных ключа. Например, рассмотрим следующий фрагмент программы:

```
/* глобальная переменная */
static pthread_key_t theKey;

/* нить A */
...
pthread_key_create(&theKey, NULL);   /* 1-й вызов */
...

/* нить B */
...
pthread_key_create(&theKey, NULL);   /* 2-й вызов */
...
```

В этом примере нити *A* и *B* выполняются параллельно, но первый вызов происходит раньше второго. При первом вызове программа создает ключ *K1* и помещает его в переменную **theKey**. При втором вызове программа создает другой ключ *K2* и помещает его в ту же переменную **theKey**, поэтому ключ *K1* уничтожается. В результате нить *A* будет работать с ключом *K2*, считая его ключом *K1*. Таких ситуаций следует избегать по следующим причинам:

- Ключ K1 потерян, поэтому отведенная под него память не будет освобождена вплоть до завершения процесса. Поскольку число ключей ограничено, их может не хватить.
- Если нить *A* использовала переменную **theKey** для хранения данных до второго вызова, эти данные были связаны с ключом K1. После второго вызова в ячейке переменной **theKey** хранится ключ K2; если теперь нить *A* попытается обратиться к своим данным, в результате она получит **NULL**.

Уникальность при создании ключей можно обеспечить следующими способами:

- С помощью средств однократной инициализации.
- Путем создания ключей до создания соответствующих нитей. Это можно реализовать, например, при работе с пулом нитей, данные которых предназначены для выполнения сходных операций. Этот пул обычно создается главной нитью.

Уникальность ключей должен обеспечивать программист. В библиотеке нитей нет средств, которые позволяли бы обнаружить повторное создание ключей.

Деструктор

С каждым ключом данных можно связать процедуру-деструктор. Если при завершении работы нити будут обнаружены какие-либо данные нити, не равные **NULL** и связанные с ключом, будет вызван деструктор для этого ключа. Он автоматически освобождает память, отведенную под данные нити, после завершения ее работы. Единственный параметр деструктора - указатель на данные нити.

Ключи данных можно использовать, например, для динамически выделенных буферов. Для освобождения буферов после завершения работы нити следует вызвать деструктор; можно применять функцию **free**:

```
pthread_key_create(&key, free);
```

Деструкторы могут иметь и более сложную структуру. Если команда **grep** с одной нитью на каждый файл хранит в области данных нитей структуру, в которую входит рабочий буфер и дескриптор файла нити, то деструктор может выглядеть следующим образом:

```
typedef struct {
        FILE *stream;
        char *buffer;
} data_t;
...

void destructor(void *data)
{
        fclose(((data_t *)data)->stream);
```

```
        free(((data_t *)data)->buffer);
        free(data);
        *data = NULL;
}
```

Деструктор можно вызвать максимум четыре раза.

Удаление ключа

Для уничтожения ключа данных нити предназначена функция **pthread_key_delete**. Функция **pthread_key_delete** не вызывает деструктор для каждой нити с данными. После уничтожения ключ данных можно повторно создать функцией **pthread_key_create**. Следовательно, функция **pthread_key_delete** оказывается полезной и при работе с несколькими ключами данных. Например, в приведенном фрагменте программы содержится бесконечный цикл:

```
/* плохой пример! */
pthread_key_t key;

while (pthread_key_create(&key, NULL))
        pthread_key_delete(key);
```

Работа с данными нитей

Для доступа к данным нитей служат функции **pthread_getspecific** и **pthread_setspecific**. Функция **pthread_getspecific** считывает значение, связанное с указанным в параметре ключом и относящееся к вызывающей нити; функция **pthread_setspecific** устанавливает это значение.

Изменение значений

С конкретным ключом должен быть связан указатель, который может указывать на данные любого типа. Как правило, данные нитей применяются для организации динамической памяти, как показано в следующем примере:

```
private_data = malloc(...);
pthread_setspecific(key, private_data);
```

При установке нового значения предыдущее значение теряется. Например, в приведенном ниже фрагменте программы значение указателя **old** теряется, а память, на которую ссылается этот указатель, превращается в "мусор":

```
pthread_setspecific(key, old);
...
pthread_setspecific(key, new);
```

Ответственность за сохранение старых указателей, т.е. за образование "мусора", несет программист. Например, можно реализовать процедуру **swap_specific** следующим образом:

```
int swap_specific(pthread_key_t key, void **old_pt, void *new)
{
        *old_pt = pthread_getspecific(key);
        if (*old_pt == NULL)
                return -1;
        else
                return pthread_setspecific(key, new);
}
```

Такой процедуры нет в библиотеке нитей, так как возвращать предыдущее значение данных нити не всегда нужно. Такая необходимость возникает, например, если данные нити являются указателями на специальные области пула памяти, выделенные главной нитью.

Работа с деструкторами

При работе с динамическими данными нитей программист должен предусмотреть для каждого вызова **pthread_key_create** вызов деструктора. Программист должен также присвоить указателю на освобожденную память значение **NULL**. В противном случае возможен вызов деструктора с недопустимыми параметрами. Пример:

```
pthread_key_create(&key, free);
...

...
private_data = malloc(...);
pthread_setspecific(key, private_data);
...

/* плохой пример! */
...
pthread_getspecific(key, &data);
free(data);
...
```

При завершении работы нити вызывается деструктор, который освобождает память, отведенную под данные нити. Поскольку значение данных - указатель на уже освобожденную память, то может возникнуть ошибка. В следующем примере ошибка исправлена:

```
/* правильный код */
...
pthread_getspecific(key, &data);
free(data);
pthread_setspecific(key, NULL);
...
```

При завершении работы нити деструктор не вызывается, поскольку все данные уже освобождены.

Работа с другими типами данных

Структура данных позволяет хранить значения, не являющиеся указателями, однако это не рекомендуется по следующим причинам:

- Преобразование указателей в скалярные типы может нарушить переносимость программ
- Значение указателя **NULL** зависит от реализации; в некоторых системах указателю **NULL** соответствует ненулевое значение.

Если вы уверены, что ваша программа никогда не будет переноситься в другую систему, то можете работать с целочисленными данными.

Понятия, связанные с данным:

“Стыковка с нитями” на стр. 444
Стыковка с нитью означает ожидание ее завершения. Эту операцию можно рассматривать как особый случай применения переменных условия.

“Разовая инициализация” на стр. 455
Некоторые библиотеки на языке C используют динамическую инициализацию, когда глобальная инициализация библиотеки выполняется при первом вызове процедуры из этой библиотеки.

“Создание сложных объектов синхронизации”
С помощью функций из библиотеки работы с нитями можно создавать более сложные объекты синхронизации.

“Список функций взаимодействия процессов нитей” на стр. 468
В этом разделе перечислены функции для взаимодействия между нитями и процессами.

Создание сложных объектов синхронизации

С помощью функций из библиотеки работы с нитями можно создавать более сложные объекты синхронизации.

Долговременные блокировки

Предусмотренные в библиотеке взаимные блокировки не отвечают требованиям активной конкуренции, поэтому их не следует устанавливать на длительное время. Для установки продолжительных блокировок используют взаимные блокировки и переменные условия, что позволяет установить блокировку на длительное время, не снижая производительность программы. Продолжительные блокировки не следует устанавливать в том случае, если разрешена отмена нитей.

Продолжительные блокировки относятся к типу данных **long_lock_t** и инициализируются с помощью функции **long_lock_init**. Функции **long_lock**, **long_trylock** и **long_unlock** выполняют операции, аналогичные **pthread_mutex_lock**, **pthread_mutex_trylock** и **pthread_mutex_unlock**.

Ниже приведен пример, демонстрирующий типичный способ применения переменных условия. Принадлежность блокировки не проверяется. Любая нить может освободить любую блокировку. Обработка ошибок и запросов на отмену не выполняется.

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int free;
        int wanted;
} long_lock_t;

void long_lock_init(long_lock_t *ll)
{
        pthread_mutex_init(&ll->lock, NULL);
        pthread_cond_init(&ll->cond);
        ll->free = 1;
        ll->wanted = 0;
}

void long_lock_destroy(long_lock_t *ll)
{
        pthread_mutex_destroy(&ll->lock);
        pthread_cond_destroy(&ll->cond);
}

void long_lock(long_lock_t *ll)
{
        pthread_mutex_lock(&ll->lock);
        ll->wanted++;
        while(!ll->free)
                pthread_cond_wait(&ll->cond);
        ll->wanted--;
        ll->free = 0;
        pthread_mutex_unlock(&ll->lock);
}

int long_trylock(long_lock_t *ll)
{
        int got_the_lock;

        pthread_mutex_lock(&ll->lock);
        got_the_lock = ll->free;
        if (got_the_lock)
                ll->free = 0;
        pthread_mutex_unlock(&ll->lock);
        return got_the_lock;
}

void long_unlock(long_lock_t *ll)
{
        pthread_mutex_lock(&ll->lock);
        ll->free = 1;
```

```
        if (ll->wanted)
                pthread_cond_signal(&ll->cond);
        pthread_mutex_unlock(&ll->lock);
}
```

Семафоры

Одно из стандартных средств синхронизации процессов - это семафоры систем UNIX. При необходимости можно реализовать семафоры для синхронизации нитей.

Семафор относится к типу данных **sema_t**, инициализируется функцией **sema_init**, а удаляется функцией **sema_destroy**. Операции ожидания и записи значения семафора выполняют функции **sema_p** и **sema_v**.

В следующем примере программы обработка ошибок не выполняется, однако при отмене нити вызываются необходимые процедуры очистки:

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int count;
} sema_t;

void sema_init(sema_t *sem)
{
        pthread_mutex_init(&sem->lock, NULL);
        pthread_cond_init(&sem->cond, NULL);
        sem->count = 1;
}

void sema_destroy(sema_t *sem)
{
        pthread_mutex_destroy(&sem->lock);
        pthread_cond_destroy(&sem->cond);
}

void p_operation_cleanup(void *arg)
{
        sema_t *sem;

        sem = (sema_t *)arg;
        pthread_mutex_unlock(&sem->lock);
}

void sema_p(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        pthread_cleanup_push(p_operation_cleanup, sem);
        while (sem->count <= 0)
                pthread_cond_wait(&sem->cond, &sem->lock);
        sem->count--;
        /*
         *  Обратите внимание: функция pthread_cleanup_pop
         *  будет выполнять функцию p_operation_cleanup
         */
        pthread_cleanup_pop(1);
}
void sema_v(sema_t *sem)
{
    pthread_mutex_lock(&sem->lock);
    if (sem->count <=0)
        pthread_cond_signal(&sem->cond);
    sem->count++;
    pthread_mutex_unlock(&sem->lock);
}
```

Счетчик задает число пользователей, которые могут получить семафор. Это значение никогда не бывает строго отрицательным; таким образом, в отличие от стандартных семафоров, здесь оно не означает число ожидающих пользователей. Такой подход иллюстрирует стандартное решение проблемы множественной активации при выполнение функции **pthread_cond_wait**. Операцию ожидания семафора можно отменить, так как в функции **pthread_cond_wait** предусмотрена точка отмены.

Блокировка для чтения и записи с приоритетом записи

Блокировка для чтения/записи с приоритетом записи предоставляет нескольким нитям возможность одновременного доступа к защищенному ресурсу для чтения и одной нити - для записи (запрещая при этом операции чтения). После снятия блокировки записывающей нитью остальные нити, ожидающие доступа на запись, имеют приоритет перед любой из считывающих нитей. Блокировки для чтения/записи с приоритетом записи обычно применяют для защиты тех ресурсов, которые чаще считывают, чем записывают.

Блокировка для чтения и записи с приоритетом записи относится к типу данных **rwlock_t** и инициализируется с помощью функции **rwlock_init**. Функция **rwlock_lock_read** блокирует ресурс для считывающей нити (или нитей), а функция **rwlock_unlock_read** - разблокирует. Функция **rwlock_lock_write** блокирует ресурс для записывающей нити, а **rwlock_unlock_write** - снимает блокировку. Важно правильно выбирать процедуры снятия блокировки для записывающих или считывающих нитей.

В следующем примере принадлежность блокировки не проверяется. Любая нить может освободить любую блокировку. Отдельные функции, в том числе **pthread_mutex_trylock**, не реализованы, и обработка ошибок не выполняется, однако при отмене операции вызываются необходимые процедуры очистки.

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t rcond;
        pthread_cond_t wcond;
        int lock_count;  /* < 0 .. блокировано для записи     */
                         /* > 0 .. блокировано для чтения */
                         /* = 0 .. никем не блокировано                  */
        int waiting_writers;  /* число ожидающих запросов на запись      */
} rwlock_t;

void rwlock_init(rwlock_t *rwl)
{
        pthread_mutex_init(&rwl->lock, NULL);
        pthread_cond_init(&rwl->wcond, NULL);
        pthread_cond_init(&rwl->rcond, NULL);
        rwl->lock_count = 0;
        rwl->waiting_writers = 0;
}

void waiting_reader_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        pthread_cleanup_push(waiting_reader_cleanup, rwl);
        while ((rwl->lock_count < 0) && (rwl->waiting_writers))
                pthread_cond_wait(&rwl->rcond, &rwl->lock);
        rwl->lock_count++;
        /*
         *  Обратите внимание: функция pthread_cleanup_pop
         *  будет выполнять функцию waiting_reader_cleanup
         */
        pthread_cleanup_pop(1);
```

```
}

void rwlock_unlock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->lock_count--;
        if (!rwl->lock_count)
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}

void waiting_writer_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        rwl->waiting_writers--;
        if ((!rwl->waiting_writers) && (rwl->lock_count >= 0))
                /*
                         * Выполняется только в случае завершения
                         */
                        pthread_cond_broadcast(&rwl->wcond);
                        pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->waiting_writers++;
        pthread_cleanup_push(waiting_writer_cleanup, rwl);
        while (rwl->lock_count)
                pthread_cond_wait(&rwl->wcond, &rwl->lock);
        rwl->lock_count = -1;
        /*
         *  Обратите внимание: функция pthread_cleanup_pop
         *  будет выполнять функцию waiting_writer_cleanup
         */
        pthread_cleanup_pop(1);
}

void rwlock_unlock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        l->lock_count = 0;
        if (!rwl->wating_writers)
                pthread_cond_broadcast(&rwl->rcond);
        else
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}
```

В отношении считывающих нитей выполняется только их подсчет. Когда их число станет равным нулю, блокировка может быть передана ожидающей записывающей нити. Блокировка может принадлежать только одной записывающей нити. После снятия блокировки будет активизирована следующая записывающая нить, если она есть. В противном случае будут активизированные все ожидающие читающие нити.

Функции блокировки можно отменить, так как они вызывают функцию **pthread_cond_wait**. В связи с этим перед вызовом этой функции регистрируются процедуры очистки.

Понятия, связанные с данным:

“Информация о нити” на стр. 457
Во многих приложениях необходимо работать с данными, относящимися к отдельным нитям.

“Обзор синхронизации” на стр. 423
Одно из главных преимуществ нитей заключается в простоте средств синхронизации.
“Список функций взаимодействия процессов нитей” на стр. 468
В этом разделе перечислены функции для взаимодействия между нитями и процессами.

Управление сигналами

Сигналы в программах с несколькими нитями представляют собой расширения сигналов, применяемых в обычных программах с одной нитью.

Управление сигналами в процессах с несколькими нитями осуществляется совместно процессом и нитями. Ниже перечислены элементы управления сигналами:

- Обработчики сигналов на уровне процесса
- Маски сигналов на уровне нитей
- Средства индивидуальной доставки каждого сигнала

Обработчики сигналов и маски сигналов

Обработчики сигналов относятся к уровню процесса. Для ожидания сигналов настоятельно рекомендуется применять функцию **sigwait**. Функцию **sigaction** применять не рекомендуется, так как список обработчиков сигналов хранится на уровне процесса, и любая нить процесса может изменять его. Если две нити указали один и тот же сигнал в обработчике сигналов, то вторая нить, вызвавшая функцию **sigaction**, переопределит параметры, заданные первой нитью; предсказать порядок, в котором будут выполняться нити, в общем случае невозможно.

Маски сигналов обслуживаются на уровне нитей. У каждой нити может быть свой набор сигналов, доставку которых она заблокирует. Для работы с маской сигналов вызывающей нити следует применять функцию **sigthreadmask**. Функцию **sigprocmask** не следует применять в программах с несколькими нитями - это может привести к непредсказуемым результатам.

Функция **pthread_sigmask** во многом схожа с функцией **sigprocmask**. Параметры и способы применения обеих функций одинаковы. При преобразовании существующей программы в версию с поддержкой библиотеки нитей функцию **sigprocmask** можно заменить на **pthread_sigmask**.

Генерация сигналов

Сигналы, относящиеся к конкретной нити, например, сигнал аппаратной неполадки, передаются в ту нить, которая послужила причиной их генерации. Сигналы, связанные с ИД процесса, ИД группы процессов или асинхронным событием (например, рабочие сигналы терминала), передаются в процесс.

- Функция **pthread_kill** передает сигнал в нить. Поскольку ИД нитей идентифицируют их только в рамках процесса, эта функция может передавать сигналы в нити только данного процесса.
- Функция **kill** (а следовательно, и команда **kill**) передает сигнал в процесс. Нить может отправить сигнал **Signal** в процесс, запустивший ее, вызвав следующую функцию:

  ```
  kill(getpid(), Signal);
  ```
- Функция **raise** не может применяться для передачи сигнала в процесс, запустивший вызывающую нить. Функция **raise** передает сигнал в вызывающую нить, как в следующем случае:

  ```
  pthread_kill(pthread_self(), Signal);
  ```

  Таким образом обеспечивается передача сигнала нити, вызвавшей функцию **raise**. Следовательно, библиотечные функции, предназначенные для применения в программах с одной нитью, можно легко перенести в систему с несколькими нитями - функция **raise** обычно применяется для передачи сигнала инициатору вызова.
- Функция **alarm** запрашивает последующую передачу сигнала в процесс; при этом аварийные состояния обрабатываются на уровне процесса. Следовательно, последняя нить, вызвавшая функцию **alarm**,

уничтожает соответствующие параметры для других нитей процесса. В программе с несколькими нитями сигнал **SIGALRM** не всегда передается в нить, вызвавшую функцию **alarm**. Может случиться, что вызывающая нить уже завершена и поэтому не может принять сигнал.

Обработка сигналов

Обработчики сигналов вызываются нитями, в которые поступили сигналы. В библиотеке нитей существуют следующие ограничения на обработчики сигналов:

- Обработчики сигналов могут вызывать функцию **longjmp** или **siglongjmp** только в том случае, если в той же нити уже выполнен вызов функции **setjmp** или **sigsetjmp**.

  Как правило, для организации ожидания сигнала в программе вызывается обработчик сигналов, который вызывает функцию **longjmp** для продолжения работы в момент вызова соответствующей функции **setjmp**. В программах с несколькими нитями такой подход неприменим, так как сигнал может быть доставлен не в ту нить, которая вызвала функцию **setjmp**: в этом случае обработчик сигналов был бы вызван из неправильной нити.

  Примечание: Вызов функции **longjmp** из обработчика сигналов может привести к непредсказуемому поведению.
- Из обработчика сигналов нельзя вызывать функции **pthread**. В результате вызова функции **pthread** из обработчика сигналов в приложении может возникнуть тупиковая ситуация.

Для того чтобы нить могла ожидать поступления асинхронного сигнала, в библиотеке нитей предусмотрена функция **sigwait**. Функция **sigwait** блокирует вызывающую нить до тех пор, пока из процесса в эту нить не поступит один из ожидаемых сигналов. Для сигналов, поступления которых ожидает функция **sigwait**, нельзя вызывать обработчик.

Обычно в программе создается отдельная нить для ожидания поступления асинхронных сигналов. Такая нить вызывает функцию **sigwait** в цикле и обрабатывает поступающие сигналы. Рекомендуется, чтобы такая нить блокировала все сигналы. Приведенный ниже фрагмент кода дает пример такой нити:

```
#include <pthread.h>
#include <signal.h>

static pthread_mutex_t mutex;
sigset_t set;
static int sig_cond = 0;

void *run_me(void *id)
{
       int sig;
       int err;
       sigset_t sigs;
       sigset_t oldSigSet;
       sigfillset(&sigs);
       sigthreadmask(SIG_BLOCK, &sigs, &oldSigSet);

       err = sigwait(&set, &sig);

       if(err)
       {
              /* обработка ошибок */
       }
       else
       {
              printf("SIGINT caught\n");
              pthread_mutex_lock(&mutex);
              sig_cond = 1;
              pthread_mutex_unlock(&mutex);
       }

       return;
```

```
}

main()
{
        pthread_t tid;

        sigemptyset(&set);
        sigaddset(&set, SIGINT);
        pthread_sigmask(SIG_BLOCK, &set, 0);

        pthread_mutex_init(&mutex, NULL);

        pthread_create(&tid, NULL, run_me, (void *)1);

        while (1)
        {
                sleep(1);
                /* или выполнить какую-либо операцию */

                pthread_mutex_lock(&mutex);
                if(sig_cond)
                {
                        /* выполнить операции для выхода */
                        return;
                }
                pthread_mutex_unlock(&mutex);

        }

}
```

Если функция **sigwait** вызвана из нескольких нитей, то при поступлении ожидаемого сигнала происходит возврат ровно из одной функции sigwait. В общем случае нельзя предсказать, какая именно нить продолжит свое выполнение. Если нить будет выполнять **sigwait**, а также управлять некоторыми другими сигналами, не выполняющих **sigwait**, то обработчики пользовательских сигналов должны блокировать сигналы sigwaiter. Следует обратить внимание, что функция **sigwait** позволяет отменить операцию.

Поскольку выделенная нить в действительности не является обработчиком сигналов, она может передать сигнал о выполнении условия в любую другую нить. Можно создать функцию **sigwait_multiple**, которая будет возобновлять выполнение всех нитей, ожидающих данного сигнала. Каждая нить, вызывающая функцию **sigwait_multiple**, может зарегистрировать свой набор сигналов. После этого нить будет ожидать изменения переменной условия. Одна нить вызывает функцию **sigwait** для всех зарегистрированных сигналов. При возврате из функции **sigwait** устанавливается соответствующее состояние и генерируются сигналы о выполнении условий. При повторном вызове функции **sigwait_multiple** ожидающий вызов функции **sigwait** будет отменен и потом возобновлен с новым набором сигналов.

Доставка сигналов

Сигнал поступает в нить, если он не должен игнорироваться. Доставка сигналов в процессах с несколькими нитями подчиняется следующим правилам:

- Если при получении сигнала следует завершить, остановить или продолжить целевую нить или процесс, то при обработке сигнала завершается, останавливается или возобновляется весь процесс (а, следовательно, все его нити). Таким образом, программы с одной нитью можно переработать в программы с несколькими нитями, не изменяя в них видимую сторону обработки сигналов.

  Рассмотрим пользовательскую команду с несколькими нитями, например, **grep**. Пользователь может запустить эту команду из оболочки, а затем попытаться прервать ее выполнение, передав соответствующий сигнал командой **kill**. Этот сигнал прервет весь процесс, в котором выполняется команда **grep**.
- Сигналы, предназначенные для конкретной нити и отправленные с помощью функции **pthread_kill** или **raise**, передаются в эту нить. Если эта нить заблокировала доставку данного сигнала, то сигнал переходит

в состояние ожидания на уровне нити, пока доставка не будет разблокирована. Если выполнение нити завершилось раньше доставки сигнала, то сигнал будет проигнорирован.

- Сигналы, предназначенные для процесса и отправленные, например, с помощью функции **kill**, передаются только одной нити процесса. Если одна или несколько нитей вызвали функцию **sigwait**, то сигнал будет передан одной из этих нитей. В противном случае сигнал передается ровно в одну нить из числа тех нитей, которые не блокировали его доставку. Если нитей, удовлетворяющих этим условиям, нет, то сигнал переходит в состояние ожидания на уровне процесса до тех пор, пока какая-либо нить не вызовет функцию **sigwait** с указанием этого сигнала или пока доставка не будет разблокирована.

Если ожидающий сигнал (на уровне нити или процесса) должен игнорироваться, то он игнорируется.

Понятия, связанные с данным:

"Порождение и завершение процессов"
Так как любой процесс содержит хотя бы одну нить, создание (т.е. порождение) и завершение процессов подразумевает создание и завершение нитей.

Список функций взаимодействия процессов нитей

В этом разделе перечислены функции для взаимодействия между нитями и процессами.

Функция	Описание
alarm	Передает в вызывающий процесс сигнал по истечении указанного периода времени.
kill и **killpg**	Передает сигнал в процесс или группу процессов.
pthread_atfork	Регистрирует программы для очистки порожденных процессов.
pthread_kill	Передает сигнал в указанную нить.
pthread_sigmask	Определяет маску сигналов нити.
raise	Передает сигнал в выполняемую нить.
sigaction, **sigvec** и **signal**	Определяет действие, выполняемое при поступлении сигнала.
sigsuspend или **sigpause**	Производит атомарное изменение в наборе блокированных сигналов и ожидает сигнала.
sigthreadmask	Определяет маску сигналов нити.
sigwait	Блокирует вызывающую нить до получения указанного сигнала.

Понятия, связанные с данным:

"Информация о нити" на стр. 457
Во многих приложениях необходимо работать с данными, относящимися к отдельным нитям.

"Создание сложных объектов синхронизации" на стр. 460
С помощью функций из библиотеки работы с нитями можно создавать более сложные объекты синхронизации.

"Порождение и завершение процессов"
Так как любой процесс содержит хотя бы одну нить, создание (т.е. порождение) и завершение процессов подразумевает создание и завершение нитей.

Порождение и завершение процессов

Так как любой процесс содержит хотя бы одну нить, создание (т.е. порождение) и завершение процессов подразумевает создание и завершение нитей.

В этом разделе рассматривается взаимодействие нитей и процессов при порождении и завершении процессов.

Порождение процессов (функция fork)

Функция **fork** вызывается в следующих случаях:

- Для создания нового потока управления в рамках уже существующей программы. AIX создает новый процесс.

- Для создания нового процесса, выполняющего другую программу. В этом случае вскоре после вызова функции **fork** вызывается одна или несколько функций **exec**.

В программе с несколькими нитями новые потоки управления создаются не функцией **fork**, а функцией **pthread_create**. Функцию **fork** следует применять только для запуска программ.

Функция **fork** дублирует не весь родительский процесс, а только ту его нить, из которой была вызвана функция; дочерний процесс будет процессом с одной нитью. Вызывающая нить родительского процесса становится главной нитью дочернего процесса, даже если в родительском процессе она не была главной. Следовательно, при возврате главной нити дочернего процесса из процедуры точки входа дочерний процесс завершается.

При дублировании родительского процесса функция **fork** дублирует и все переменные синхронизации, включая их состояния. Это может привести к нарушению целостности ресурсов в случае, когда нить дочернего процесса установила взаимную блокировку, а затем завершилась.

Настоятельно рекомендуется использовать функцию **fork** только для запуска новых программ и сразу же вызывать одну из функций **exec** в дочернем процессе, порожденном функцией **fork**.

Обработчики fork

При работе с библиотеками, обеспечивающими поддержку нескольких нитей, перечисленных выше мер предосторожности недостаточно. Программе не всегда известно о том, что применяется библиотека с поддержкой нитей, поэтому в ней может вызываться любое количество библиотечных функций между вызовами функций **fork** и **exec**. Прежние однонитевые версии программы не всегда могут соответствовать новым требованиям библиотеки нитей.

С другой стороны, многонитевым библиотекам необходимы средства защиты своего внутреннего состояния при вызове функции fork на случай, если из дочернего процесса будет вызвана какая-либо функция библиотеки. Эта проблема особенно актуальна для многонитевых библиотек ввода-вывода, так как практически всегда между вызовами функций **fork** и **exec** приходится вызывать какую-либо из библиотечных функций для перенаправления ввода-вывода.

С помощью функции **pthread_atfork** можно защитить библиотеки с поддержкой нескольких нитей от последствий вызова функции **fork** из приложений. Эта функция также реализует стандартный механизм защиты приложений с несколькими нитями от последствий вызова **fork** из библиотеки или из самих приложений.

Функция **pthread_atfork** регистрирует обработчики fork, которые необходимо вызвать до и после вызова функции **fork**. Эти обработчики выполняются в той нити, из которой вызывается функция **fork**. Предусмотрены следующие обработчики fork:

Функция	Описание
Подготовительный обработчик	Подготовительный обработчик вызывается непосредственно перед началом обработки функции **fork**.
Обработчик для родительского процесса	Этот обработчик вызывается в родительском процессе сразу же после завершения обработки функции **fork**.
Обработчик для дочернего процесса	Этот обработчик вызывается в дочернем процессе сразу же после завершения обработки функции **fork**.

Завершение процесса

Подготовительные обработчики вызываются в порядке стека (LIFO), а обработчики для родительского и дочернего процессов - в порядке простой очереди (FIFO). Благодаря этому в программах можно сохранить исходный порядок блокировки.

При завершении процесса с помощью явного или неявного вызова функции **exit**, **atexit** или **_exit** завершаются все нити процесса. Не вызываются ни обработчики очистки, ни деструкторы данных для нитей.

Примечание: Процедура **unatexit** аннулирует регистрацию функций, которые были ранее зарегистрированы с помощью процедуры **atexit**. Если указанная функция будет найдена, она будет удалена из списка функций, вызываемых при нормальном завершении работы программы.

Это связано с тем, что процесс завершается целиком, вместе со всеми своими нитями, так что очищается и освобождается вся память.

Понятия, связанные с данным:

“Сведения о нитях и процессах” на стр. 407
Нить - это независимый поток управления, работающий в том же адресном пространстве, что и остальные независимые потоки управления в рамках процесса.

“Управление сигналами” на стр. 465
Сигналы в программах с несколькими нитями представляют собой расширения сигналов, применяемых в обычных программах с одной нитью.

“Список функций взаимодействия процессов нитей” на стр. 468
В этом разделе перечислены функции для взаимодействия между нитями и процессами.

Необязательные компоненты библиотеки работы с нитями

В данном разделе описаны расширенные атрибуты нитей, взаимных блокировок и переменных условий.

В стандарте библиотек нитей POSIX реализация некоторых компонентов объявлена необязательной. Все функции, определенные с помощью API библиотеки нитей, доступны всегда. Некоторые функции могут быть не реализованы. Приложения могут вызывать нереализованные функции, но такие функции всегда будут возвращать код ошибки **ENOSYS**.

Атрибуты стека

Для каждой нити выделяется собственный стек. Управление стеком зависит от реализации. Таким образом, приведенная ниже информация применима только к AIX, хотя подобные функции могут существовать и в других системах.

Стек выделяется динамически при создании нити. С помощью расширенных атрибутов нити пользователь может управлять размером стека и задавать его адрес. Приведенная ниже информация не относится к начальной нити, создаваемой системой.

Размер стека

С помощью размера стека можно изменять значение атрибута **stacksize** в объекте атрибутов нити. Этот атрибут указывает минимальный размер области памяти, отведенной под стек для вновь созданной нити.

В системе AIX определен атрибут **stacksize**. Если этот необязательный компонент реализован, то доступны следующие атрибуты и функции:

- Атрибут **stacksize** из объекта атрибутов нити
- Функция **pthread_attr_getstacksize** возвращает значение атрибута
- Функция **pthread_attr_setstacksize** задает значение атрибута

Значение атрибута **stacksize** по умолчанию равно 96 Кб. Минимальное значение атрибута **stacksize** по умолчанию равно 16 Кб. Если указанное значение меньше минимального, то будет применяться минимальное значение.

В библиотеке поддержки нитей AIX для каждой новой нити выделяется блок данных, называемый *пользовательской областью нити*. Этот блок состоит из следующих компонентов:

- *Красная зона*, защищенная от чтения и записи, предназначена для обнаружения переполнения стека. В программах с большими страницами красная зона не предусмотрена.
- Стек по умолчанию.
- Структура pthread.
- Структура thread.
- Структура атрибутов thread.

Примечание: Необходимо различать пользовательскую область нити, описанную здесь, и структуру **uthread**, применяемую в ядре AIX. Пользовательская область нити доступна только в пользовательском режиме и управляет ей исключительно библиотека нитей, в то время как структура **uthread** существует только в среде ядра.

Адрес стека компонента POSIX

С адресом стека связано значение атрибута **stackaddr** в объекте атрибутов нити. Этот атрибут указывает место в памяти, отведенное под стек для вновь созданной нити.

Если этот необязательный компонент реализован, то доступны следующие атрибуты и функции:
- Атрибут **stackaddr** задает адрес стека, выделяемого нити.
- Функция **pthread_attr_getstackaddr** возвращает значение атрибута.
- Функция **pthread_attr_setstackaddr** задает значение атрибута.

Если адрес не указан, то нити выделяется стек с произвольным адресом. Если стек необходимо определить по конкретному адресу, то вы можете воспользоваться атрибутом **stackaddr**. Например, если нужен стек очень большого размера, вы можете указать для него адрес из свободного сегмента.

Если адрес стека был указан при вызове функции **pthread_create**, то система попытается выделить стек по этому адресу. Если сделать это невозможно, то функция **pthread_create** вернет значение **EINVAL**. Функция **pthread_attr_setstackaddr** возвращает ошибку только в том случае, если указанный адрес стека не попадает в адресное пространство.

Планирование приоритета компонента POSIX

Планирование приоритета позволяет управлять планированием нитей. Если эта функция отключена, то параметры планирования всех нитей в процессе наследуются от самого процесса. Если же она включена, то каждая нить имеет свой набор параметров планирования. Для нитей с локальной областью действия с параметрами планирования работает планировщик библиотеки на уровне процесса, в то время как параметры планирования нитей с глобальной областью действия обрабатывает планировщик ядра на системном уровне.

Если этот необязательный компонент реализован, то доступны следующие атрибуты и функции:
- Атрибут **inheritsched** из объекта атрибутов нити
- Атрибут **schedparam** из объекта атрибутов нити и из самой нити
- Атрибут **schedpolicy** из объекта атрибутов нити и из самой нити
- Атрибут **contention-scope** из объекта атрибутов нити и из самой нити
- Функции **pthread_attr_getschedparam** и **pthread_attr_setschedparam**
- Функция **pthread_getschedparam**

Проверка наличия необязательного компонента

Проверить наличие необязательных компонентов можно как на этапе компиляции, так и на этапе выполнения. Переносимые программы должны проверять наличие необязательных компонентов до начала работы с ними, чтобы не приходилось изменять исходный код программ при переносе в другую систему.

Проверка во время компиляции

Если окажется, что какой-либо необязательный компонент недоступен, то компиляцию можно прервать так:

```
#ifndef _POSIX_THREAD_ATTR_STACKSIZE
#error "Необходимо наличие компонента POSIX - Размер стека"
#endif
```

В файле заголовков **pthread.h** определяются также символы, которые могут применяться другими файлами заголовков или программами. Эти символы перечислены ниже:

_POSIX_REENTRANT_FUNCTIONS
Означает, что необходимы реентерабельные функции.

_POSIX_THREADS
Означает реализацию библиотеки нитей.

Проверка во время выполнения

Для проверки наличия необязательных компонентов в системе на этапе выполнения программы можно применять функцию **sysconf**. Она оказывается весьма полезной при переносе программ в двоично-совместимую систему, например, в систему под управлением другой версии AIX.

Ниже приведен список символьных констант, связанных со всеми необязательными компонентами. Эти символьные константы необходимо передать в функцию **sysconf** в параметре *Name*. Константы определены в файле заголовка **unistd.h**.

Адрес стека
_SC_THREAD_ATTR_STACKADDR

Размер стека
_SC_THREAD_ATTR_STACKSIZE

Планирование приоритета
_SC_THREAD_PRIORITY_SCHEDULING

Наследование приоритета
_SC_THREAD_PRIO_INHERIT

Защита приоритета
_SC_THREAD_PRIO_PROTECT

Совместное выполнение процессов
_SC_THREAD_PROCESS_SHARED

Наличие основных компонентов можно проверить, вызвав функцию **sysconf** со следующими значениями параметра *имя*:

_SC_REENTRANT_FUNCTIONS
Означает, что необходимы реентерабельные функции.

_SC_THREADS
Означает реализацию библиотеки нитей.

Совместное выполнение процессов

В AIX и большинстве систем UNIX допускается использование несколькими процессами одного пространства данных, называемого *общей памятью*. Атрибуты совместного выполнения процессов, задаваемые для переменных условий и взаимных блокировок, позволяют размещать эти объекты в общей памяти для синхронизации нитей различных процессов. Однако стандартного интерфейса управления общей памятью не существует, поэтому опция POSIX для общих процессов в AIX не реализована.

Типы данных библиотеки нитей

Для работы с библиотекой нитей определены следующие типы данных определены. Определения этих типов данных могут быть различными в различных системах.

pthread_t
Идентифицирует нить.

pthread_attr_t
Идентифицирует объект атрибутов нити.

pthread_cond_t
Идентифицирует условную переменную.

pthread_condattr_t
Идентифицирует объект атрибутов условной переменной.

pthread_key_t
Идентифицирует ключ данных для конкретной нити.

pthread_mutex_t
Идентифицирует взаимную блокировку.

pthread_mutexattr_t
Идентифицирует объект атрибутов взаимной блокировки.

pthread_once_t
Идентифицирует объект разовой инициализации.

Ограничения и значения по умолчанию

В библиотеке нитей есть несколько ограничений и значений по умолчанию, зависящих от реализации. Для улучшения переносимости программ можно считывать эти ограничения и значения по умолчанию в виде символьных констант:

- В процессе не может быть более 512 нитей. Максимальное количество нитей можно получить на этапе компиляции с помощью символьной константы **PTHREAD_THREADS_MAX**, определенной в файле заголовка **pthread.h**. Если приложение скомпилировано с флагом **-D_LARGE_THREADS**, то максимальное число нитей в одном процессе равно 32767.
- Минимальный размер стека для одной нити составляет 8 Кб. По умолчанию размер стека равен 96 Кб. Минимальный размер стека можно получить на этапе компиляции с помощью символьной константы **PTHREAD_STACK_MIN**, определенной в файле заголовка **pthread.h**.

  Примечание: Максимальный размер стека составляет 256 Мб, что равно размеру сегмента. Это ограничение можно получить с помощью символьной константы **PTHREAD_STACK_MAX** в файле заголовка **pthread.h**.
- Максимальное значение этого параметра - 508. Это значение можно получить на этапе компиляции с помощью символьной константы **PTHREAD_KEYS_MAX**, определенной в файле заголовка **pthread.h**.

Значения атрибутов по умолчанию

Значения атрибутов нитей по умолчанию определены в файле заголовка **pthread.h** и связаны со следующими символьными константами:

- Значение символьной константы **DEFAULT_DETACHSTATE** по умолчанию равно **PTHREAD_CREATE_DETACHED** и задает значение атрибута **detachstate** по умолчанию.
- Значение символьной константы **DEFAULT_JOINABLE** по умолчанию равно **PTHREAD_CREATE_JOINABLE** и задает значение объединяемого состояния по умолчанию.
- Значение символьной константы **DEFAULT_INHERIT** по умолчанию равно **PTHREAD_INHERIT_SCHED** и задает значение атрибута **inheritsched** по умолчанию.

- Значение символьной константы **DEFAULT_PRIO** по умолчанию равно 1 и задает значение по умолчанию для поля `sched_prio` атрибута **schedparam**.
- Значение символьной константы **DEFAULT_SCHED** по умолчанию равно **SCHED_OTHER** и задает значение атрибута нитей **schedpolicy** по умолчанию.
- Значение символьной константы **DEFAULT_SCOPE** по умолчанию равно **PTHREAD_SCOPE_LOCAL** и задает значение атрибута **contention-scope** по умолчанию.

Понятия, связанные с данным:

“Защита нитей и библиотеки поддержки нитей в AIX” на стр. 412
В этом разделе рассмотрены библиотеки поддержки нитей в AIX.

Список дополнительных функций для работы с нитями

В этом разделе перечислены дополнительные функции для работы с нитями.

Функция	Описание
pthread_attr_getstackaddr	Возвращает значение атрибута stackaddr объекта атрибутов нити.
pthread_attr_getstacksize	Возвращает значение атрибута stacksize объекта атрибутов нити.
pthread_attr_setstackaddr	Устанавливает значение атрибута stackaddr объекта атрибутов нити.
pthread_attr_setstacksize	Устанавливает значение атрибута stacksize объекта атрибутов нити.
pthread_condattr_getpshared	Возвращает значение общего атрибута процессов объекта атрибутов условия.
pthread_condattr_setpshared	Устанавливает значение общего атрибута процессов в объекте атрибутов условия.
pthread_getspecific	Возвращает данные нити, связанные с определенным ключом.
pthread_key_create	Создает ключ для данных нити.
pthread_key_delete	Удаляет ключ для данных нити.
pthread_mutexattr_getpshared	Возвращает значение общего атрибута процессов из объекта атрибутов взаимной блокировки.
pthread_mutexattr_setpshared	Устанавливает значение общего атрибута процессов в объекте атрибутов взаимной блокировки.
pthread_once	Выполняет процедуру в процессе только один раз.
PTHREAD_ONCE_INIT	Инициализирует управляющую структуру однократной синхронизации.
pthread_setspecific	Задает данные нити, связанные с определенным ключом.

Поддерживаемые интерфейсы

В системах AIX символьные константы **_POSIX_THREADS**, **_POSIX_THREAD_ATTR_STACKADDR**, **_POSIX_THREAD_ATTR_STACKSIZE** и **_POSIX_THREAD_PROCESS_SHARED** определены всегда.

Следовательно, поддерживаются следующие интерфейсы работы с нитями.

Интерфейсы POSIX

Ниже приведен список интерфейсов POSIX:

- pthread_atfork
- pthread_attr_destroy
- pthread_attr_getdetachstate
- pthread_attr_getschedparam
- pthread_attr_getstacksize
- pthread_attr_getstackaddr
- pthread_attr_init
- pthread_attr_setdetachstate
- pthread_attr_setschedparam
- pthread_attr_setstackaddr
- pthread_attr_setstacksize

- pthread_cancel
- pthread_cleanup_pop
- pthread_cleanup_push
- pthread_detach
- pthread_equal
- pthread_exit
- pthread_getspecific
- pthread_join
- pthread_key_create
- pthread_key_delete
- pthread_kill
- pthread_mutex_destroy
- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_mutexattr_destroy
- pthread_mutexattr_getpshared
- pthread_mutexattr_init
- pthread_mutexattr_setpshared
- pthread_once
- pthread_self
- pthread_setcancelstate
- pthread_setcanceltype
- pthread_setspecific
- pthread_sigmask
- pthread_testcancel
- pthread_cond_broadcast
- pthread_cond_destroy
- pthread_cond_init
- pthread_cond_signal
- pthread_cond_timedwait
- pthread_cond_wait
- pthread_condattr_destroy
- pthread_condattr_getpshared
- pthread_condattr_init
- pthread_condattr_setpshared
- pthread_create
- sigwait

Интерфейсы Single UNIX Specification версии 2

Ниже приведен список интерфейсов Single UNIX Specification версии 2:

- pthread_attr_getguardsize
- pthread_attr_setguardsize
- pthread_getconcurrency

- pthread_mutexattr_gettype
- pthread_mutexattr_settype
- pthread_rwlock_destroy
- pthread_rwlock_init
- pthread_rwlock_rdlock
- pthread_rwlock_tryrdlock
- pthread_rwlock_trywrlock
- pthread_rwlock_unlock
- pthread_rwlock_wrlock
- pthread_rwlockattr_destroy
- pthread_rwlockattr_getpshared
- pthread_rwlockattr_init
- pthread_rwlockattr_setpshared
- pthread_setconcurrency

В системах AIX символьная константа **_POSIX_THREAD_SAFE_FUNCTIONS** определена всегда. Следовательно, всегда поддерживаются следующие интерфейсы:

- asctime_r
- ctime_r
- flockfile
- ftrylockfile
- funlockfile
- getc_unlocked
- getchar_unlocked
- getgrgid_r
- getgrnam_r
- getpwnam_r
- getpwuid_r
- gmtime_r
- localtime_r
- putc_unlocked
- putchar_unlocked
- rand_r
- readdir_r
- strtok_r

Перечисленные ниже интерфейсы не поддерживаются в AIX; их идентификаторы есть в системе, но попытка обратиться к ним всегда приводит к ошибке с errno = ENOSYS:

- pthread_mutex_getprioceiling
- pthread_mutex_setprioceiling
- pthread_mutexattr_getprioceiling
- pthread_mutexattr_getprotocol
- pthread_mutexattr_setprioceiling
- pthread_mutexattr_setprotocol

Интерфейсы без поддержки нитей

Библиотека libc.a (стандартные функции):

- advance
- asctime
- brk
- catgets
- chroot
- compile
- ctime
- cuserid
- dbm_clearerr
- dbm_close
- dbm_delete
- dbm_error
- dbm_fetch
- dbm_firstkey
- dbm_nextkey
- dbm_open
- dbm_store
- dirname
- drand48
- ecvt
- encrypt
- endgrent
- endpwent
- endutxent
- fcvt
- gamma
- gcvt
- getc_unlocked
- getchar_unlocked
- getdate
- getdtablesize
- getgrent
- getgrgid
- getgrnam
- getlogin
- getopt
- getpagesize
- getpass
- getpwent
- getpwnam
- getpwuid
- getutxent
- getutxid
- getutxline

- getw
- getw
- gmtime
- l64a
- lgamma
- localtime
- lrand48
- mrand48
- nl_langinfo
- ptsname
- putc_unlocked
- putchar_unlocked
- pututxline
- putw
- rand
- random
- readdir
- re_comp
- re_exec
- regcmp
- regex
- sbrk
- setgrent
- setkey
- setpwent
- setutxent
- sigstack
- srand48
- srandom
- step
- strerror
- strtok
- ttyname
- ttyslot
- wait3

Перечисленные ниже интерфейсы AIX не обеспечивают защиту нитей.

Библиотека libc.a (специальные функции AIX):
- endfsent
- endttyent
- endutent
- getfsent
- getfsfile
- getfsspec
- getfstype

- getttyent
- getttynam
- getutent
- getutid
- getutline
- pututline
- setfsent
- setttyent
- setutent
- utmpname

Библиотека **libbsd.a**:
- timezone

Библиотеки **libm.a** и **libmsaa.a**:
- gamma
- lgamma

Библиотеки, ни одна функция в которых не обеспечивает защиту нитей:
- libPW.a
- libblas.a
- libcur.a
- libcurses.a
- libplot.a
- libprint.a

Интерфейсы **ctermid** и **tmpnam** не обеспечивают защиту нитей при передаче аргумента NULL.

В многонитевой программе не рекомендуется выполнять функцию **setlocale()** одновременно из нескольких нитей, если одна из нитей вызывает функцию **setlocale()** из подпрограммы инициализации модуля.

Примечание: В некоторых системах ряд функций может быть реализован в виде макросов. Не используйте адреса функций для работы с нитями.

Написание реентерабельных программ и программ с поддержкой нитей

В процессах с одной нитью только один поток управления, и обеспечивать реентерабельность кода или поддержку нитей не требуется. В процессах с несколькими нитями одни и те же функции и ресурсы могут использоваться несколькими потоками управления одновременно.

поэтому для обеспечения целостности ресурсов код должен быть реентерабельным и предусматривать поддержку нитей.

Оба понятия - реентерабельность и поддержка нитей - связаны со способом работы функций с ресурсами. Однако это различные свойства: функция может обладать одним из них, обоими или ни одним.

В этом разделе содержится информация о написании реентерабельных программ и программ с поддержкой нитей. В нем не обсуждаются вопросы повышения эффективности нитей, т.е. оптимального распараллеливания потоков. Добиться такой эффективности можно только за счет удачного алгоритма. Существующие программы с одной нитью можно преобразовать в эффективные программы с несколькими нитями, однако для этого потребуется полностью переработать алгоритм и переписать программу заново.

Реентерабельность

Реентерабельная функция не может ни хранить статические данные в промежутках между вызовами, ни возвращать указатель на статические данные. Все данные передаются из вызывающей функции. Реентерабельная функция не может вызывать нереентерабельную функцию.

Реентерабельную функцию часто (но не всегда) можно определить по внешнему интерфейсу и по характеру применения. Например, функция **strtok** нереентерабельна, так как она хранит строку, разбиваемую на маркеры. Функция **ctime** также нереентерабельна, поскольку она возвращает указатель на статические данные, изменяемые при каждом вызове.

Поддержка нитей

Функция с поддержкой нитей обеспечивает защиту общих ресурсов от одновременного доступа путем установки блокировок. Определить функцию с поддержкой нитей по внешнему интерфейсу невозможно, так как поддержка нитей реализуется только на уровне алгоритма.

В языке C локальные переменные динамически помещаются в стек. Поэтому любая функция, не работающая со статическими данными и другими общими ресурсами, очевидно обеспечивает поддержку нитей, как это показано в следующем примере:

```
/* функция с поддержкой нитей */
int diff(int x, int y)
{
        int delta;

        delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
}
```

Применение глобальных данных не обеспечивает поддержку нитей. Глобальные данные необходимо обрабатывать в рамках какой-либо одной нити или инкапсулировать для сериализации доступа к ним. Нить может считывать код ошибки из другой нити. В AIX у каждой нити собственное значение **errno**.

Создание реентерабельных функций

В большинстве случаев для преобразования нереентерабельной функции в реентерабельную достаточно переделать только интерфейс. Нереентерабельные функции не могут применяться в нескольких нитях одновременно. Кроме того, в некоторых нереентерабельных функциях невозможно обеспечить поддержку нитей.

Возврат данных

Многие нереентерабельные функции возвращают указатели на статические данные. Избежать этого можно следующими способами:

- Возвращать динамические данные. В этом случае освобождение памяти выполняется в вызывающей функции. Преимущество такого подхода заключается в том, что не нужно переделывать интерфейс. Однако при этом не гарантируется совместимость с остальными программами: при вызове измененной функции из программы с одной нитью программа не освободит память.
- Использовать память, выделенную в вызывающей функции. Это рекомендуемый подход, хотя он и требует переработки интерфейса.

Например, функция **strtoupper**, преобразующая строку к верхнему регистру, может выглядеть так:

```
/* нереентерабельная функция */
char *strtoupper(char *string)
{
        static char buffer[MAX_STRING_SIZE];
        int index;

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

Эта функция нереентерабельная (и не обеспечивает поддержку нитей). Если выбран первый способ преобразования функции в реентерабельную, то ее исправленный код может выглядеть так:

```
/* реентерабельная функция (неудачный вариант) */
char *strtoupper(char *string)
{
        char *buffer;
        int index;

        /* необходима проверка наличия ошибок! */
        buffer = malloc(MAX_STRING_SIZE);

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

Более удачный вариант требует переработки интерфейса. Вызывающая функция должна выделить память для строки ввода и строки вывода, как в следующем фрагменте программы:

```
/* реентерабельная функция (более удачный вариант) */
char *strtoupper_r(char *in_str, char *out_str)
{
        int index;

        for (index = 0; in_str[index]; index++)
        out_str[index] = toupper(in_str[index]);
        out_str[index] = 0

        return out_str;
}
```

Стандартные нереентерабельные библиотеки языка C были переработаны именно способом выделения памяти в вызывающей функции.

Хранение данных в промежутках между вызовами

Хранить данные в промежутках между вызовами нельзя, так как вызовы могут осуществляться из разных нитей. Если какие-либо данные (например, рабочий буфер или указатель) необходимо сохранить после возврата из функции, то их следует хранить в вызывающей функции.

Рассмотрим следующий пример. Пусть функция должна возвращать следующую строчную букву данной строки, причем сама строка указывается только при первом вызове функции, как в функции **strtok**. При достижении конца строки функция должна возвращать 0. Код такой функции может выглядеть так:

```
/* нереентерабельная функция */
char lowercase_c(char *string)
{
        static char *buffer;
        static int index;
```

```
        char c = 0;

        /* сохранение строки при первом вызове */
        if (string != NULL) {
                buffer = string;
                index = 0;
        }

        /* поиск строчной буквы */
        for (; c = buffer[index]; index++) {
                if (islower(c)) {
                        index++;
                        break;
                }
        }
        return c;
}
```

h

Эта функция нереентерабельна. Для того чтобы она стала реентерабельной, необходимо, чтобы статические данные, т.е. переменная **index**, хранились в вызывающей функции. Переработанный код функции может выглядеть так:

```
/* реентерабельная функция */
char reentrant_lowercase_c(char *string, int *p_index)
{
        char c = 0;

        /* инициализация не выполняется - она уже выполнена в вызывающей функции */

        /* поиск строчной буквы */
        for (; c = string[*p_index]; (*p_index)++) {
                if (islower(c)) {
                        (*p_index)++;
                        break;
                  }
        }
        return c;
}
```

Изменились и интерфейс функции, и способ ее применения. Вызывающая функция должна предоставлять строку при каждом вызове этой функции и инициализировать переменную index нулем до первого вызова, как в следующем фрагменте программы:

```
char *my_string;
char my_char;
int my_index;
...
my_index = 0;
while (my_char = reentrant_lowercase_c(my_string, &my_index)) {
        ...
}
```

Создание функций с поддержкой нитей

В программах с несколькими нитями все функции, вызываемые из нескольких нитей, должны обеспечивать поддержку нитей. Однако существуют способы вызова в таких программах функций без поддержки нитей. Нереентерабельные функции обычно не являются функциями с поддержкой нитей, но после переработки в реентерабельные часто становятся таковыми.

Блокировка общих ресурсов

Функции, работающие со статическими данными или с любыми другими общими ресурсами, например, файлами и терминалами, должны для обеспечения поддержки нитей сериализовать доступ к этим ресурсам с помощью блокировок. Например, следующая функция не обеспечивает поддержку нитей:

```
/* функция без поддержки нитей */
int increment_counter()
{
        static int counter = 0;

        counter++;
        return counter;
}
```

Для того чтобы обеспечить поддержку нескольких нитей, статическую переменную **counter** необходимо защитить с помощью статической блокировки, как показано в следующем примере:

```
/* псевдокод функции с поддержкой нитей */
int increment_counter();
{
        static int counter = 0;
        static lock_type counter_lock = LOCK_INITIALIZER;

        pthread_mutex_lock(counter_lock);
        counter++;
        pthread_mutex_unlock(counter_lock);
        return counter;
}
```

В приложении с несколькими нитями, работающем с библиотекой нитей, для сериализации доступа к общим ресурсам следует применять взаимные блокировки. При работе с независимыми библиотеками может потребоваться использовать их вне контекста нитей и потому устанавливать блокировки других типов.

Способы безопасного применения функций без поддержки нитей

Существует несколько специальных способов безопасного применения функций без поддержки нитей в программах, в которых эти функции вызываются из нескольких нитей. Эти способы могут пригодиться, в частности, при подключении библиотеки без поддержки нитей к программе с несколькими нитями - например, для тестирования, либо если версия этой библиотеки с поддержкой нитей пока не разработана. Реализация этих способов представляет собой достаточно сложную задачу, так как требуется осуществить сериализацию обработки данной функции или даже группы функций. Существуют следующие способы:

- Первый способ - глобальная блокировка всей библиотеки. Блокировка устанавливается при каждом обращении к библиотеке (т.е. при каждом вызове библиотечной функции или при каждом обращении к библиотечной глобальной переменной). Недостаток такого подхода заключается в возможном снижении производительности, так как в любой момент времени к каким бы то ни было средствам библиотеки сможет обращаться только одна нить. Описанный ниже способ можно применять только в случае, если обращения к библиотеке редки, или же временно.

  ```
  /* псевдокод! */

  lock(library_lock);
  library_call();
  unlock(library_lock);

  lock(library_lock);
  x = library_var;
  unlock(library_lock);
  ```

- Второй способ - блокировка каждого отдельного компонента библиотеки (т.е. каждой функции и каждой глобальной переменной) или группы компонентов. Этот вариант значительно более трудоемок, но не приводит к снижению производительности. Так как эти способы должны применяться только в приложениях, но не в библиотеках, для защиты библиотек можно применять взаимные блокировки.

```
/* псевдокод! */

lock(library_moduleA_lock);
library_moduleA_call();
unlock(library_moduleA_lock);

lock(library_moduleB_lock);
x = library_moduleB_var;
unlock(library_moduleB_lock);
```

Реентерабельные библиотеки с поддержкой нитей

Реентерабельные библиотеки с поддержкой нитей применяются во многих параллельных (а также асинхронных) средах программирования, а не только при программировании нитей. Рекомендуется применять и создавать только функции, обладающие свойствами реентерабельности и поддержки нитей.

Работа с библиотеками

Часть библиотек, входящих в комплект поставки Базовой операционной системы AIX, обеспечивают поддержку нитей. В текущей версии AIX это следующие библиотеки:

- Стандартная библиотека C (**libc.a**)
- Библиотека, обеспечивающая совместимость с Berkeley (**libbsd.a**)

Некоторые стандартные функции C, например, **ctime** и **strtok**, нереентерабельны. Имена реентерабельных версий этих функций отличаются суффиксом **_r** (знак подчеркивания и буква *_r*).

При написании программ с несколькими нитями следует применять реентерабельные функции. Например, следующий фрагмент кода:

```
token[0] = strtok(string, separators);
i = 0;
do {
        i++;
        token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

в программе с несколькими нитями необходимо переработать так:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
        i++;
        token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

К библиотеке без поддержки нитей может обращаться только одна нить программы. Ответственность за соблюдение этого правила несет разработчик программы; нарушение этого правила может привести к непредсказуемым результатам и даже сбою программы.

Преобразование библиотек

В этом разделе описаны основные этапы преобразования уже существующей библиотеки в реентерабельную библиотеку с поддержкой нитей. Содержимое этого раздела относится только к библиотекам языка C.

- Выявление экспортируемых глобальных переменных. Эти переменные обычно определяются в файлах заголовка с помощью ключевого слова **export**. Экспортируемые глобальные переменные необходимо инкапсулировать, т.е. сделать их закрытыми (объявить их в исходном коде библиотеки с помощью ключевого слова **static**) и создать для них операции чтения и записи.

- Выявление статических переменных и других общих ресурсов. Эти переменные обычно определяются с помощью ключевого слова **static**. Установка защиты посредством блокировок для всех общих ресурсов. Количество блокировок влияет на производительность библиотеки. Для инициализации блокировок можно воспользоваться функцией однократной инициализации.
- Выявление нереентерабельных функций и преобразование их в реентерабельные. Дополнительная информация приведена в разделе Создание реентерабельных функций.
- Выявление функций без поддержки нитей и преобразование их в функции с поддержкой нитей. Дополнительная информация приведена в разделе Создание функций с поддержкой нитей.

Понятия, связанные с данным:

“Разовая инициализация” на стр. 455
Некоторые библиотеки на языке C используют динамическую инициализацию, когда глобальная инициализация библиотеки выполняется при первом вызове процедуры из этой библиотеки.

Информация, связанная с данной:

admin

cdc

delta

get

prs

sccsdiff

sccsfile

Создание программ с несколькими нитями

Создание программ с несколькими нитями аналогично разработке программ, состоящих из нескольких процессов. Процесс разработки программы включает в себя компиляцию и отладку исходного кода.

Компиляция программ с несколькими нитями

Этот раздел посвящен компиляции программ с несколькими нитями. В нем приведена информация по следующим вопросам:

- Необходимый заголовочный файл
- Вызов компилятора для создания программы с несколькими нитями.

Заголовочный файл

Все прототипы процедур, макроопределения и прочие определения, необходимые для применения библиотеки нитей, собраны в заголовочном файле **pthread.h**, который расположен в каталоге **/usr/include**. Заголовочный файл **pthread.h** следует включить во все исходные файлы, которые работают с библиотекой нитей.

pthread.h включает заголовочный файл **unistd.h**, который, в свою очередь, содержит определения:

_POSIX_REENTRANT_FUNCTIONS
: Указывает, что все функции должны иметь возможность повторного вхождения. В некоторых файлах заголовков этот символ применяется для определения дополнительных функций с возможностью повторного вхождения, таких, как процедура **localtime_r**.

_POSIX_THREADS
: Обозначает API нитей POSIX. Это имя применяется для проверки доступности API нитей POSIX. В других API нитей определения некоторых процедур и макросов могут быть заданы по-другому.

Файл **pthread.h** также включает **errno.h**, в котором глобальная переменная **errno** переопределена как переменная нити. Следовательно, идентификатор **errno** в многопоточных программах не является значением типа **l-value**.

Вызов компилятора

Для компиляции программы с несколькими нитями необходимо вызвать компилятор языка C с помощью одной из следующих команд:

xlc_r
Вызывает компилятор для языка по умолчанию **ansi**

cc_r
Вызывает компилятор для языка по умолчанию **extended**

Эти команды гарантируют, чтобы выбранные опции и библиотеки будут соответствовать стандарту Single UNIX Specification версии 2. Стандарт POSIX Threads Specification 1003.1c является подмножеством стандарта Single UNIX Specification версии 2.

При вызове команд **xlc_r** и **cc_r** к программе автоматически подключаются следующие библиотеки:

libpthreads.a
Библиотека нитей

libc.a
Стандартная библиотека C

Например, приведенная ниже команда компилирует исходный файл программы с несколькими нитями на языке C (**foo.c**) и создает исполняемый файл **foo**:

```
cc_r -o foo foo.c
```

Вызов компилятора для проекта 7 стандарта POSIX 1003.1c

AIX поддерживает приложения, созданные в соответствии с проектом 7. Разработчикам рекомендуется обновить свои приложения с нитями в соответствии с современным стандартом.

Для компиляции программы с несколькими нитями на уровне проекта 7 нужно вызвать компилятор языка C с помощью одной из следующих команд:

xlc_r7
Вызывает компилятор для языка по умолчанию **ansi**

cc_r7
Вызывает компилятор для языка по умолчанию **extended**

При вызове команд **xlc_r7** и **cc_r7** к программе автоматически подключаются следующие библиотеки:

libpthreads_compat.a
Библиотека нитей, совместимая с проектом 7

libpthreads.a
Библиотека нитей

libc.a
Стандартная библиотека C

Для обеспечения совместимости исходного кода укажите директиву компилятора **_AIX_PTHREADS_D7**. Библиотеки должны быть подключены в следующем порядке: **libpthreads_compat.a**, **libpthreads.a** и **libc.a**. Большинство пользователей могут не вникать в эти подробности, так как эти команды обеспечивают выполнение указанных требований. Эта информация предназначена для тех, кто не располагает последней версией компилятора AIX.

Преобразование приложений, соответствующих проекту 7, к стандарту &Symbol.unixspec;

Между окончательным вариантом стандарта и проектом 7 существуют следующие различия:

- Незначительно отличаются значения **errno**. В окончательном варианте стандарта в ситуации, когда указанная нить не найдена, возвращается значение **ESRCH**. В проекте 7 в той же ситуации часто возвращалось значение **EINVAL**.
- По умолчанию нить создается как *подключаемая*. Это существенное изменение, на которое следует обратить внимание. В противном случае может возникнуть утечка памяти в программе.
- Параметр планирования нити по умолчанию - *scope*.
- Функция **pthread_yield** заменена на функцию **sched_yield**.
- Внесены небольшие изменения в стратегии планирования, связанные с взаимными блокировками.

Объем памяти, необходимой для программ с несколькими нитями

В одном процессе AIX может быть создано до 32768 нитей. Каждая нить занимает определенную часть адресного пространства процесса, поэтому фактически максимальное число нитей процесса зависит от применяемой модели памяти и объема адресного пространства процесса, необходимого для других целей. Фрагмент адресного пространства, отведенный нити, содержит стек, защищенную область памяти и небольшую область памяти для внутреннего использования. Размер стека можно задать с помощью функции **pthread_attr_setstacksize**, а размер защищенной области памяти - с помощью функции **pthread_attr_setguardsize**.

Примечание: Гибкое ограничение размера стека, заданное с помощью команды **ulimit** *–s*, применимо только для стека главной нити приложения.

В следующей таблице указано максимальное число нитей, которые может создать в рамках 32-разрядного процесса простая программа, создающая в цикле новые нити с атрибутом NULL и не выполняющая никаких других действий. В реальных программах максимальное число нитей зависит от объема памяти, используемой в других целях. Максимальное число нитей 64-разрядного процесса зависит от ограничения **ulimit**. В связи с этим применение модели большого объема данных может привести к уменьшению максимального числа нитей.

Модель данных	-bmaxdata	Максимальное число нитей
Небольшой объем данных	н/д	1084
Большой объем данных	0x10000000	2169
Большой объем данных	0x20000000	4340
Большой объем данных	0x30000000	6510
Большой объем данных	0x40000000	8681
Большой объем данных	0x50000000	10852
Большой объем данных	0x60000000	13022
Большой объем данных	0x70000000	15193
Большой объем данных	0x80000000	17364

Переменная среды **NUM_SPAREVP** позволяет управлять числом резервных виртуальных процессоров, обслуживаемых библиотекой. Изменять эту переменную не нужно. В некоторых ситуациях приложения, которые используют лишь несколько мегабайт памяти, могут снизить накладные затраты на управление памятью путем присвоения переменной **NUM_SPAREVP** меньшего значения. Обычно значение соответствует числу CPU системы или пиковому числу нитей процесса. Изменение этой переменной не влияет на производительность процесса. Значение по умолчанию равно 256.

Примечание: Переменная среды **NUM_SPAREVP** доступна только в AIX 5.1.

Пример программы с несколькими нитями

Следующая короткая программа с несколькими нитями показывает приветствие на английском и французском языках. Эта программа должна быть откомпилирована с помощью **cc_r** или **xlc_r**. F

```
#include <pthread.h>    /* первый включаемый файл - pthread.h */
#include <stdio.h>      /* поддержка функции printf() */
#include <unistd.h>     /* поддержка функции sleep()          */

void *Thread(void *string)
{
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        sleep(5);

        /* Обычно процедура exit не используется.
           Объяснение приведено ниже. */
        exit(0);
}
```

Исходная нить (в которой выполняется функция **main**) создает две нити. Обе нити используют одну процедуру точки входа (**Thread**), но с разными параметрами. В качестве параметра передается указатель на выводимую строку.

Отладка программ с несколькими нитями

Для отладки программ с несколькими нитями предусмотрены следующие инструменты:

- Для отладки приложений можно воспользоваться командой **dbx**. Для просмотра объектов, связанных с нитями, в ней предусмотрено несколько подкоманд, в том числе **attribute**, **condition**, **mutex** и **thread**.
- Для отладки расширений ядра и драйверов устройств можно применять отладчик ядра. Эта программа не предназначена для работы с пользовательскими нитями. Главным образом она работает с нитями ядра. Некоторые подкоманды поддерживают несколько нитей ядра и несколько процессоров, в том числе:
  - Подкоманда **cpu**, изменяющая текущий процессор
  - Подкоманда **ppd**, показывающая структуры данных препроцессора
  - Подкоманда **thread**, показывающая записи таблицы нитей
  - Подкоманда **uthread**, показывающая структуру **uthread** нити

  Дополнительная информация о программе отладки ядра содержится в *Kernel Extensions and Device Support Programming Concepts*.

Требования к файлу дампа программ с несколькими нитями

По умолчанию процессы не создают полный дамп. Если при отладке необходимо просмотреть данные из общего сегмента памяти, например, из стека нити, то нужно создать полный дамп. Для создания полного дампа переключитесь на пользователя root и введите следующую команду:

```
chdev -l sys0 -a fullcore=true
```

Размер файла дампа пропорционален числу нитей. Файл дампа включает в себя содержимое стека нити, размер которого можно изменить с помощью функции **pthread_attr_setstacksize**. Для нитей, созданных с атрибутом NULL, в файл дампа дополнительно добавляется 128 Кб информации (в случае 32-разрядного процесса) или 256 Кб (в случае 64-разрядного процесса).

Понятия, связанные с данным:

“Защита нитей и библиотеки поддержки нитей в AIX” на стр. 412
В этом разделе рассмотрены библиотеки поддержки нитей в AIX.

“Создание нитей” на стр. 413
Создание нитей отличается от создания процессов тем, что между нитями не существует "родственных" отношений (предок-потомок).

“Планирование нитей” на стр. 447
В библиотеке нитей предусмотрен набор функций для планирования нитей и управления планированием.

“Создание программ с несколькими нитями” на стр. 485
Создание программ с несколькими нитями аналогично разработке программ, состоящих из нескольких процессов. Процесс разработки программы включает в себя компиляцию и отладку исходного кода.

Разработка программ с несколькими нитями, которые проверяют и изменяют объекты библиотеки нитей

Библиотека отладки нитей (**libpthdebug.a**) содержит ряд функций, позволяющих приложениям проверять и изменять объекты библиотеки нитей.

Эта библиотека применяется как для 32-разрядных, так и для 64-разрядных приложений. Она обеспечивает поддержку нескольких нитей. Библиотека отладки нитей содержит 32-разрядный общий объект.

Библиотека отладки нитей предоставляет приложениям доступ к информации, хранящейся в библиотеке нитей. К такой информации относятся сведения о нитях, их атрибутах, взаимных блокировках, атрибутах блокировок, условных переменных, атрибутах условных переменных, блокировках чтения и записи, атрибутах этих блокировок, а также сведения о состоянии библиотеки нитей.

Примечание: Все данные (адреса и регистры) возвращаются функциями библиотеки в 64-разрядном формате как при работе с 64-разрядными, так и при работе с 32-разрядными приложениями. Преобразование этих значений в 32-разрядный формат для 32-разрядных приложений должно выполняться самими приложениями. При отладке 32-разрядных приложений старшие 32 разряда адреса или регистра отбрасываются.

Библиотека отладки нитей не сообщает сведения о взаимных блокировках и их атрибутах, переменных условия, атрибутах этих переменных, блокировках чтения-записи и атрибутах этих блокировок, для которых параметр pshared равен **PTHREAD_PROCESS_SHARED**.

Инициализация

Приложение должно инициализировать библиотеку отладки нитей для каждого процесса с несколькими нитями. После загрузки процесса с несколькими нитями в нем должна быть вызвана функция **pthdb_sessison_init**. Библиотека отладки нитей поддерживает только один сеанс для каждого процесса. Приложение должно присвоить уникальный идентификатор пользователя и передать его в функцию **pthdb_session_init**, которая, в свою очередь, должна присвоить уникальный идентификатор сеансу. Этот идентификатор передается в качестве первого параметра всем остальным функциям **библиотеки отладки нитей**, за исключением **pthdb_session_pthreaded**. Когда библиотека отладки нитей запускает функцию обратного вызова, она передает приложению выделенный им идентификатор пользователя. Функция

pthdb_session_init проверяет список функций обратного вызова, предоставляемых приложением, и инициализирует структуры данных сеанса. Кроме того, эта функция устанавливает флаги сеанса. Приложение должно передать флаг **PTHDB_FLAG_SUSPEND** функции **pthdb_session_init**. Полный список флагов приведен в описании функции **pthdb_session_setflags**.

Функции обратного вызова

Библиотека отладки нитей применяет функции обратного вызова для получения и сохранения данных, а также для управления памятью приложений. Для приложения требуются следующие функции обратного вызова:

read_data
: Получает информацию из объекта **pthread library**

alloc Выделяет память в библиотеке отладки нитей

realloc Изменяет объем памяти, выделенный в библиотеке отладки нитей

dealloc Освобождает память, выделенную в библиотеке отладки нитей

Дополнительно для приложения предусмотрены следующие функции обратного вызова:

read_regs
: Применяется только функциями **pthdb_pthread_context** и **pthdb_pthread_setcontext**

write_data
: Применяется только функцией **pthdb_pthread_setcontext**

write_regs
: Применяется только функцией **pthdb_pthread_setcontext**

Функция обновления

При каждой остановке приложения после инициализации сеанса необходимо вызывать функцию **pthdb_session_update**. Эта функция устанавливает или обновляет списки нитей, атрибутов нитей, взаимных блокировок, атрибутов взаимных блокировок, переменных условия, атрибутов этих переменных, блокировок чтения/записи, атрибутов этих блокировок, ключей отдельных нитей и активных ключей. Память для списков выделяется с помощью функций обратного вызова.

Функции работы с контекстом

Для получения контекстной информации служит функция **pthdb_pthread_context**, а для ее задания - функция **pthdb_pthread_setcontext**. Функция **pthdb_pthread_context** считывает контекст пользовательской нити из структуры данных нити ядра или пользовательской нити. Эта структура данных расположена в адресном пространстве приложения. Если пользовательская нить не связана с нитью ядра, контекстная информация считывается из библиотеки нитей. Если пользовательская нить связана с нитью ядра, необходимая информация считывается из приложения с помощью функций обратного вызова. При этом приложение должно определить, работает ли нить ядра в режиме ядра или пользовательском режиме, и выдать информацию для соответствующего режима.

Если пользовательская нить связана с нитью ядра, находящейся в режиме ядра, то нельзя считать полную информацию о контексте для пользовательского режима, так как ядро хранит ее компоненты в разных местах. Часть этой информации можно получить с помощью функции **getthrds**, так как она всегда сохраняет стек пользовательского режима. Приложение может получить доступ к этой информации, проверив значение **thrdsinfo64.ti_scount**. Если оно отлично от нуля, значит в структуре **thrdsinfo64.ti_ustk** находится стек пользовательского режима. С помощью стека пользовательского режима можно определить регистр адреса команды (IAR) и страницы функций обратного вызова, но нельзя узнать значения других регистров. Определение структуры **thrdsinfo64** содержится в файле **procinfo.h**.

Функции списка

Библиотека отладки нитей управляет списками нитей, атрибутов нитей, взаимных блокировок, атрибутов взаимных блокировок, переменных условия, атрибутов переменных условия, блокировок чтения/записи,

атрибутов блокировок чтения/записи, ключей отдельных нитей и активных ключей, представленных в виде ссылок соответствующих типов. Функции вида **pthdb_*объект*** возвращают указатель на следующий элемент соответствующего списка, где *объект* может принимать значение **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** или **key**. Если список пуст или достигнут конец списка, возвращается значение **PTHDB_INVALID_*ОБЪЕКТ***, где *ОБЪЕКТ* - это **PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR** или **KEY**.

Функции работы с полями

Дополнительную информацию об объекте можно получить с помощью соответствующих функций работы с элементами объекта. Они имеют вид **pthdb_*объект_поле***, где *объект* - это **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** или **key**, а *поле* - имя поля с дополнительной информацией об объекте.

Настройка сеанса

Приложение может изменить флаги сеанса с помощью функции **pthdb_session_setflags**. Эти флаги определяют количество регистров, считываемых и записываемых при работе с контекстом.

Текущие флаги сеанса можно получить с помощью функции **pthdb_session_flags**.

Завершение сеанса

В конце сеанса необходимо освободить память, выделенную под структуры данных сеанса, и удалить данные сеанса. Это можно сделать с помощью функции **pthdb_session_destroy**, которая освобождает память посредством функции обратного вызова. Эта функция освобождает всю память, которая была получена функциями **pthdb_session_init** и **pthdb_session_update**.

Пример подключения к библиотеке отладки нитей

В приведенном ниже примере программы продемонстрировано, каким образом приложение может подключиться к библиотеке отладки нитей:

```
/* директивы include */

#include &lt;thread.h>
#include &lt;ys/pthdebug.h>

...

int my_read_data(pthdb_user_t user, pthdb_symbol_t symbols[],int count)
{
  int rc;

  rc=memcpy(buf,(void *)addr,len);
  if (rc==NULL) {
    fprintf(stderr,&odq;Error message\n&cdq;);
    return(1);
  }
  return(0);
}
int my_alloc(pthdb_user_t user, size_t len, void **bufp)
{
  *bufp=malloc(len);
  if(!*bufp) {
    fprintf(stderr,&odq;Error message\n&cdq;);
    return(1);
  }
  return(0);
}
int my_realloc(pthdb_user_t user, void *buf, size_t len, void **bufp)
{
  *bufp=realloc(buf,len);
```

```
  if(!*bufp) {
    fprintf(stderr,"Сообщение об ошибке\n");
    return(1);
  }
  return(0);
}
int my_dealloc(pthdb_user_t user,void *buf)
{
  free(buf);
  return(0);
}

status()
{
  pthdb_callbacks_t callbacks =
                    {  NULL,
                       my_read_data,
                       NULL,
                       NULL,
                       NULL,
                       my_alloc,
                       my_realloc,
                       my_dealloc,
                       NULL
                    };

  ...

  rc=pthread_suspend_others_np();
  if (rc!=0)
    обработка ошибок

  if (not initialized)
    rc=pthdb_session_init(user,exec_mode,PTHDB_SUSPEND|PTHDB_REGS,callbacks,
                          &session);
    if (rc!=PTHDB_SUCCESS)
       обработка ошибок

  rc=pthdb_session_update(session);
  if (rc!=PTHDB_SUCCESS)
        обработка ошибок

   получить информацию об объекте pthread с помощью функций списка
   и функций обработки полей

  ...

  rc=pthread_continue_others_np();
  if (rc!=0)
    обработка ошибок
}

...

main()
{
  ...
}
```

Разработка отладчиков для программ с несколькими нитями

Библиотека отладки нитей (**libpthdebug.a**) содержит набор функций, предназначенных для отладки программ, в которых используется библиотека нитей.

Эта библиотека предназначена для отладки 32-разрядных и 64-разрядных приложений с несколькими нитями. Она может применяться только для процессов, поддерживающих отладку. Кроме того, с ее

помощью можно получить информацию о нитях ее собственного приложения. На основе библиотеки можно создать отладчик с несколькими нитями, предназначенный для приложений с несколькими нитями. Библиотека **libpthreads.a** поддерживает работу с несколькими нитями и допускает использование отладчиков с несколькими нитями. Библиотека отладки нитей содержит 32-разрядный общий объект.

Отладчики, использующие утилиту ptrace, должны применять 32-разрядную версию библиотеки, так как утилита ptrace не поддерживается в 64-разрядном режиме. Отладчики, использующие утилиту /proc, могут применять и 32-, и 64-разрядную версию этой библиотеки.

Библиотека отладки нитей предоставляет отладчикам доступ к информации, хранящейся в библиотеке нитей. В число этой информации входят сведения о нитях, их атрибутах, взаимных блокировках, атрибутах блокировок, условных переменных, атрибутах условных переменных, блокировках чтения и записи, атрибутах этих блокировок, а также сведения о состоянии библиотеки pthread. Кроме того, в этой библиотеке содержатся средства для управления работой нитей.

Примечание: Все данные (адреса и регистры) возвращаются функциями библиотеки в 64-разрядном формате как при работе с 64-разрядными, так и при работе с 32-разрядными приложениями. Преобразование этих значений в 32-разрядный формат для 32-разрядных приложений должно выполняться отладчиком. При отладке 32-разрядных приложений старшие 32 разряда адреса или регистра отбрасываются.

Библиотека отладки нитей не сообщает сведения о взаимных блокировках и их атрибутах, переменных условия и их атрибутах, блокировках чтения/записи и их атрибутах, для которых параметр pshared равен PTHREAD_PROCESS_SHARED.

Инициализация

Отладчик должен инициализировать сеанс библиотеки отладки нитей для каждого процесса отладки. Это невозможно сделать до инициализации библиотеки нитей. С помощью функции **pthdb_session_pthreaded** отладчик может узнать, была ли инициализирована библиотека нитей. Функция **pthdb_session_pthreaded** проверяет, инициализирована ли библиотека нитей. Если да, то функция возвращает значение `PTHDB_SUCCESS`. Если нет, она возвращает значение `PTHDB_NOT_PTHREADED`. В обоих случаях она возвращает имя функции, с помощью которой можно установить точку прерывания для немедленного получения уведомления об инициализации библиотеки нитей. Таким образом, функция **pthdb_session_pthreaded** позволяет узнать об инициализации библиотеки нитей следующими способами:

- Отладчик может вызывать эту функцию при каждой остановке в процессе отладки для того чтобы узнать, активизированы ли нити в отлаживаемой программе.
- Отладчик может вызвать эту функцию один раз, а затем установить точку прерывания для получения уведомления об активизации нитей процесса, если эти нити еще не активизированы.

После инициализации библиотеки нитей отладчик должен вызвать функцию **pthdb_session_init** для инициализации сеанса отладки. Библиотека отладки нитей поддерживает только один сеанс для каждого процесса отладки. Отладчик должен присвоить уникальный ИД пользователя и передать его в функцию **pthdb_session_init**, которая, в свою очередь, должна присвоить уникальный идентификатор сеансу. Этот идентификатор передается в качестве первого параметра всем остальным функциям библиотеки отладки нитей, за исключением **pthdb_session_pthreaded**. Когда библиотека отладки нитей запускает функцию обратного вызова, она передает отладчику выделенный им идентификатор пользователя. Функция **pthdb_session_init** проверяет список функций обратного вызова, предоставляемых отладчиком, и инициализирует структуры данных сеанса. Кроме того, эта функция устанавливает флаги сеанса. См. описание функции **pthdb_session_setflags** в *Technical Reference: Base Operating System and Extensions, Volume 1*.

Функции обратного вызова

Библиотека отладки нитей применяет функции обратного вызова для выполнения следующих действий:

- Получения адресов и данных

- Записи данных
- Передачи функций управления памятью отладчику
- При отладке самой библиотеки отладки нитей

Функция обновления

При каждой остановке отладчика после инициализации сеанса необходимо вызывать функцию **pthdb_session_update**. Эта функция устанавливает или обновляет списки нитей, атрибутов нитей, взаимных блокировок, атрибутов взаимных блокировок, переменных условия, атрибутов этих переменных, блокировок чтения/записи, атрибутов этих блокировок, ключей отдельных нитей и активных ключей. Память для списков выделяется с помощью функций обратного вызова.

Функции блокирования и освобождения

Отладчики должны поддерживать блокирование и освобождение нитей по следующим причинам:

- Для пошаговой отладки нити необходимо, чтобы отладчик заблокировал остальные нити.
- Для отладки группы нитей отладчик должен заблокировать все нити, не входящие в группу.

Для блокирования и разблокирования нитей применяются следующие функции:

- Функция **pthdb_pthread_hold** устанавливает для атрибута *состояние блокировки* нити значение `блокирована`.
- Функция **pthdb_pthread_unhold** устанавливает для атрибута *состояние блокировки* нити значение `разблокирована`.

  Примечание: Функции **pthdb_pthread_hold** и **pthdb_pthread_unhold** должны применяться всегда, независимо от того, существует ли для нити соответствующая нить ядра.
- Функция **pthdb_pthread_holdstate** возвращает значение атрибута *состояние блокировки* нити.
- Функция **pthdb_session_committed** возвращает имя функции, которая будет вызвана после фиксации всех изменений состояния блокировки нити. В эту функцию можно добавить точку прерывания, чтобы сообщить отладчику о фиксации изменений.
- Функция **pthdb_session_stop_tid** передает в **библиотеку отладки нитей** ИД нити (TID), вызвавшей остановку отладчика. Далее эта информация передается в **библиотеку нитей**.
- Функция **pthdb_session_commit_tid** поочередно возвращает ИД нитей ядра, работу которых нужно возобновить для фиксации изменений. Эту функцию нужно вызывать в цикле до тех пор, пока не будет получено значение `PTHDB_INVALID_TID`. Если список нитей ядра пуст, для фиксации не нужно возобновлять работу каких-либо нитей.

Отладчик может узнать о том, что все изменения состояния блокировки зафиксированы, следующими способами:

- Перед запуском операции фиксации (после возобновления работы всех нитей, ИД которых были возвращены функцией **pthdb_session_commit_tid**) отладчик может вызвать функцию **pthdb_session_committed**, определить с ее помощью имя функции и установить точку прерывания. (Этот способ можно использовать только один раз за время работы процесса).
- Перед запуском операции фиксации отладчик может вызвать функцию **pthdb_session_stop_tid**, передав в нее ИД нити, вызвавшей остановку отладчика. После фиксации библиотека нитей проверяет, что остановлена нить с тем же самым ИД, что и до фиксации.

Для блокирования нитей перед отладкой группы нитей или пошаговой отладкой одной нити нужно выполнить следующие действия:

1. С помощью функций **pthdb_pthread_hold** и **pthdb_pthread_unhold** укажите, какие нити должны быть блокированы, а какие - разблокированы.
2. Выберите способ проверки фиксации всех изменений состояния блокировки.

3. С помощью функции **pthdb_session_commit_tid** определите список ИД нитей, выполнение которых необходимо продолжить для фиксации изменений.
4. Возобновите работу нитей из этого списка, а также нити, вызвавшей остановку отладчика.

Функция **pthdb_session_continue_tid** позволяет отладчику получить список нитей ядра, выполнение которых нужно продолжить перед переходом к пошаговой отладке нити или отладке группы нитей. Эту функцию нужно вызывать в цикле до тех пор, пока не будет получено значение `PTHDB_INVALID_TID`. Если список нитей ядра не пуст, отладчик должен возобновить работу указанных в нем нитей вместе с отлаживаемыми нитями. Остановка и возобновление работы отлаживаемой нити выполняется отладчиком. Отлаживаемая нить - это нить, вызвавшая остановку отладчика.

Функции работы с контекстом

Для получения контекстной информации служит функция **pthdb_pthread_context**, а для ее задания - функция **pthdb_pthread_setcontext**. Функция **pthdb_pthread_context** считывает контекст пользовательской нити из структуры данных нити ядра или пользовательской нити. Эта структура данных расположена в адресном пространстве процесса отладки. Если пользовательская нить не связана с нитью ядра, контекстная информация считывается из библиотеки нитей. Если пользовательская нить связана с нитью ядра, необходимая информация считывается из отладчика с помощью функций обратного вызова. При этом отладчик должен определить, работает ли нить ядра в режиме ядра или пользовательском режиме, и выдать информацию для соответствующего режима.

Если пользовательская нить связана с нитью ядра, находящейся в режиме ядра, то нельзя считать полную информацию о контексте для пользовательского режима, так как ядро хранит ее компоненты в разных местах. Часть этой информации можно получить с помощью функции **getthrds**, так как она всегда сохраняет стек пользовательского режима. Отладчик может получить доступ к этой информации, проверив значение **thrdsinfo64.ti_scount**. Если оно отлично от нуля, значит в структуре **thrdsinfo64.ti_ustk** находится стек пользовательского режима. С помощью стека пользовательского режима можно определить регистр адреса команды (IAR) и страницы функций обратного вызова, но нельзя узнать значения других регистров. Определение структуры **thrdsinfo64** содержится в файле **procinfo.h**.

Функции списка

Библиотека отладки нитей управляет списками нитей, атрибутов нитей, взаимных блокировок, атрибутов взаимных блокировок, переменных условия, атрибутов переменных условия, блокировок чтения/записи, атрибутов блокировок чтения/записи, ключей отдельных нитей и активных ключей, представленных в виде ссылок соответствующих типов. Функции вида **pthdb_*объект*** возвращают ссылку на следующий элемент соответствующего списка, где *объект* может принимать одно из следующих значений: **pthread**, **attr**, **mutex**, **mutexattr**, **cond**, **condattr**, **rwlock**, **rwlockattr** или **key**. Если список пуст, либо был достигнут конец списка, возвращается значение **PTHDB_INVALID_*объект***, где *объект* - это одно из следующих значений: **PTHREAD** , **ATTR**, **MUTEX**, **MUTEXATTR**, **COND**, **CONDATTR**, **RWLOCK**, **RWLOCKATTR** или **KEY**.

Функции работы с полями

Подробную информацию об объекте можно получить с помощью соответствующих функций работы с элементами объекта. Они имеют вид **pthdb_*объект_поле***, где ***объект*** - это значение **pthread**, **attr**, **mutex**, **mutexattr**, **cond**, **condattr**, **rwlock**, **rwlockattr** или **key**, а ***поле*** - это имя поля подробной информации об объекте.

Настройка сеанса

Отладчик может изменить флаги сеанса с помощью функции **pthdb_session_setflags**. Эти флаги задают количество регистров, считываемых и записываемых при работе с контекстом, а также управляют печатью отладочной информации.

Текущие флаги сеанса можно получить с помощью функции **pthdb_session_flags**.

Завершение сеанса

При завершении сеанса отладки необходимо освободить память, выделенную под структуры данных сеанса, и удалить данные сеанса. Это можно сделать с помощью функции **pthdb_session_destroy**, которая освобождает память посредством функции обратного вызова. Эта функция освобождает всю память, которая была получена функциями **pthdb_session_init** и **pthdb_session_update**.

Пример применения функций блокирования и разблокирования

Приведенный ниже пример программы демонстрирует применение функций блокирования и разблокирования в работе отладчика:

```
/* директивы include */

#include <sys/pthdebug.h>

main()
{
    tid_t stop_tid; /* нить, вызывающая остановку процесса */
    pthdb_user_t user = <уникальное значение отладчика>;
    pthdb_session_t session; /* <уникальное библиотечное значение> */
    pthdb_callbacks_t callbacks = <функции обратного вызова>;
    char *pthreaded_symbol=NULL;
    char *committed_symbol;
    int pthreaded = 0;
    int pthdb_init = 0;
    char *committed_symbol;

    /* fork/exec или подключение к отлаживаемой программе */

    /* отлаживаемая программа применяет функции ptrace()/ptracex() с опцией PT_TRACE_ME */

    while (/* ожидание события */)
    {
      /* отладчик ждет активизации отлаживаемой программы */

      if (pthreaded_symbol==NULL) {
        rc = pthdb_session_pthreaded(user, &callbacks, pthreaded_symbol);
        if (rc == PTHDB_NOT_PTHREADED)
        {
            /* установка точки прерывания для pthreaded_symbol */
        }
        else
          pthreaded=1;
      }
      if (pthreaded == 1 && pthdb_init == 0) {
          rc = pthdb_session_init(user, &session, PEM_32BIT, flags, &callbacks);
          if (rc)
              /* обработка ошибки и выход */
          pthdb_init=1;
      }

      rc = pthdb_session_update(session)
      if ( rc != PTHDB_SUCCESS)
 /* обработка ошибки и выход */

      while (/* считывание команд отладчика */)
      {
          switch (/* команда отладчика */)
          {
              ...
              case DB_HOLD:
                  /* независимо от того, связана нить с нитью ядра или нет */
                  rc = pthdb_pthread_hold(session, pthread);
                  if (rc)
                      /* обработка ошибки и выход */
              case DB_UNHOLD:
                  /* независимо от того, связана нить с нитью ядра или нет */
                  rc = pthdb_pthread_unhold(session, pthread);
```

```
                if (rc)
                    /* обработка ошибки и выход */
            case DB_CONTINUE:
                /* если нам не нужно блокировать нить до конца */
                /* процесса */
                if (pthreaded)
                {
                    /* отладчик должен обработать список любой длины */
                    struct pthread commit_tids;
                    int commit_count = 0;
                    /* отладчик должен обработать список любой длины */
                    struct pthread continue_tids;
                    int continue_count = 0;

       rc = pthdb_session_committed(session, committed_symbol);
       if (rc != PTHDB_SUCCESS)
    /* обработка ошибки */
            /* установка точки прерывания для committed_symbol */

                    /* получение всех ИД нитей, необходимых для фиксации */
                    /* всех операций блокировки/освобождения */
                    do
                    {
                        rc = pthdb_session_commit_tid(session,
                                        &commit_tids.th[commit_count++]);
                        if (rc != PTHDB_SUCCESS)
                            /* обработка ошибки и выход */
                    } while (commit_tids.th[commit_count - 1] != PTHDB_INVALID_TID);

                    /* остановка обработки нити, вызвавшей останов */
                    /* процесса, с помощью функции stop_park */

       if (commit_count > 0) {
                      rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                          &commit_tids);
                      if (rc)
                          /* обработка ошибки и выход */

                      /* ожидание остановки процесса */
       }

                    /* получение всех ИД нитей, необходимых для продолжения */
                    /* обработки нужных нитей */
                    do
                    {
                        rc = pthdb_session_continue_tid(session,
                                        &continue_tids.th[continue_count++]);
                         if (rc != PTHDB_SUCCESS)
                            /* обработка ошибки и выход */
                    } while (continue_tids.th[continue_count - 1] != PTHDB_INVALID_TID);

                    /* добавление нужных нитей к списку continue_tids */

                    /* остановка нити, вызвавшей остановку */
                    /* процесса, если она не нужна */

                    rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                           &continue_tids);
                    if (rc)
                        /* обработка ошибки и выход */
                }
            case DB_EXIT:
  rc = pthdb_session_destroy(session);
  /* завершение очистки */
  exit(0);
            ...
          }
      }

    }
    exit(0);
}
```

Достоинства нитей

Производительность программ с несколькими нитями выше, чем у обычных параллельных программ, использующих несколько процессов. В многопроцессорных системах нити дают дополнительный выигрыш в производительности.

Управление нитями

Управление нитями, включающее создание нитей и контроль за их выполнением, требует меньше ресурсов, чем управление процессами. Например, для создания нити требуется только частная область данных, размер которой обычно составляет 64 Кб, и два системных вызова. Для создания процесса необходимо гораздо больше ресурсов - при этом копируется все адресное пространство родительского процесса.

Библиотечные API для работы с нитями также проще использовать, чем библиотеку управления процессами. Для создания нити требуется вызвать только функцию **pthread_create**.

Взаимодействие нитей

Взаимодействие нитей намного эффективнее и проще взаимодействия процессов. Все нити одного процесса используют одно адресное пространство, поэтому создавать для них общую память не требуется. Для защиты общих данных от одновременного использования применяются взаимные блокировки или другой механизм синхронизации.

Функции синхронизации, доступные в библиотеке работы с нитями, позволяют создавать гибкие и надежные средства синхронизации. Эти средства могут применяться вместо традиционных средств взаимодействия процессов, например, очередей сообщений. Для организации взаимодействия между нитями можно использовать каналы.

Многопроцессорные системы

В многопроцессорных системах несколько нитей могут одновременно выполняться на нескольких CPU. Вследствие этого программы с несколькими нитями могут выполняться намного быстрее, чем в однопроцессорных системах. Они также могут выполняться быстрее программ с несколькими процессами, так как нити требуют меньше ресурсов и создают меньше служебной информации. Например, переключение нитей в одном процессе может выполняться быстрее, особенно в модели библиотеки M:N, в которой контекстные переключатели часто пропускаются. Основное же преимущество нитей заключается в том, что программа с несколькими нитями может работать в однопроцессорной системе, но для переноса ее в многопроцессорную систему повторная компиляция не потребуется.

Ограничения

Программирование с использованием нитей применяется для реализации алгоритмов параллельной обработки с помощью нескольких независимых модулей. Однако в некоторых случаях вместо программ с несколькими нитями приходится применять программы с несколькими процессами.

Многие идентификаторы, ресурсы, признаки состояния и ограничения операционной системы определены на уровне процессов и, таким образом, используются совместно всеми нитями процесса. Например, ИД групп и пользователей, а также связанные с ними права доступа обрабатываются на уровне процесса. В программах, в которых различным модулям требуется присвоить различные идентификаторы, приходится использовать несколько процессов, вместо одного процесса с несколькими нитями. Другими примерами могут служить атрибуты файловой системы, такие как текущий рабочий каталог, а также состояние и максимальное число открытых файлов. Если для выполнения задачи предпочтительна независимая обработка этих атрибутов, то программы с несколькими нитями могут оказаться неподходящими. Например, программа с несколькими процессами допускает открытие каждым процессом большого числа файлов без согласования с другими процессами.

Понятия, связанные с данным:

“Защита нитей и библиотеки поддержки нитей в AIX” на стр. 412
В этом разделе рассмотрены библиотеки поддержки нитей в AIX.

Информация о программах lex и yacc

Для того чтобы программа могла обрабатывать вводимые данные в интерактивном или пакетном режиме, необходимо разработать процедуру или дополнительную программу, которая будет обрабатывать входной поток. Если ввод сложный, то требуется дополнительный модуль для разбиения ввода на фрагменты, интерпретируемые программой как отдельные, независимые данные.

Для разработки таких программ ввода можно применять команды **lex** и **yacc**.

Команда **lex** создает лексический анализатор, который считывает ввод и разбивает его на лексемы, такие как числа, буквы и операторы. Лексемы определяются правилами грамматики, описанными в файле спецификаций **lex**. Команда **yacc** создает синтаксический анализатор, который обрабатывает ввод с помощью лексем, созданных командой **lex** и хранящихся в файле спецификаций **lex**, и выполняет указанные действия, например, отмечает синтаксические ошибки. С помощью этих команд можно создать лексический и синтаксический анализаторы для разбора входных и форматирования выходных данных.

Информация, связанная с данной:

printf

ed

ex

sed

yacc

Создание лексического анализатора с помощью команды lex

Команда **lex** помогает создать программу на языке C, которая может получать поток символов и преобразовывать его в действия программы.

Для использования команды **lex** необходим файл спецификаций, который содержит:

Расширенные регулярные выражения
Наборы символов, распознаваемые созданным лексическим анализатором.

Операторы действий
Компоненты программы на C, определяющие действия, которые выполняются для распознаваемых расширенных регулярных выражений.

Более подробная информация о формате и назначении этого файла приведена в описании команды **lex** в книге *Справочник по командам, том 3*.

Команда **lex** создает программу на языке C, которая может анализировать входной поток на основе информации, заданной в файле спецификаций. Команда **lex** создает программу в файле **lex.yy.c**. Если полученная программа применяется для распознавания простого ввода, в котором команде соответствует одно слово, вы можете скомпилировать файл **lex.yy.c** и получить исполняемую программу-лексический анализатор:

```
cc lex.yy.c -ll
```

Если программа должна распознавать более сложный синтаксис, потребуется разработать синтаксический анализатор который будет отвечать за обработку ввода совместно с лексическим анализатором.

Файл вывода **lex.yy.c** можно перенести в любую систему, в которой есть компилятор C, поддерживающий библиотечные функции **lex**.

Откомпилированный лексический анализатор выполняет следующие операции:

- Считывает входной поток символов.
- Копирует входной поток в выходной поток.
- Разбивает входной поток на подстроки, соответствующие расширенным регулярным выражениям, описанным в файле спецификаций **lex**.
- Выполняет действия, определенные для распознаваемых анализатором расширенных регулярных выражений. Эти действия определяются фрагментами программ на C, заданными в файле спецификаций **lex**. Такие фрагменты могут вызывать внешние по отношению к ним действия или функции.

Лексический анализатор, создаваемый командой **lex**, применяет метод анализа, называемый *детерминированным конечным автоматом*. Этот метод задает ограниченное число состояний лексического анализатора и определяет правила, согласно которым устанавливается состояние анализатора.

Автомат применяет опережающий просмотр более чем на один или два символа вперед. Предположим, например, что в файле спецификаций **lex** определено два правила: одно распознает строку `ab`, а другое - строку `abcdefg`. Если лексический анализатор получает на вход строку `abcdefh`, он считывает все символы до конца строки, а затем определяет, что строка не совпадает с `abcdefg`. После этого он возвращается к правилу, соответствующему строке `ab`, определяет, что оно соответствует части входной строки, обрабатывает его и начинает анализ оставшейся строки - `cdefh`.

Компиляция лексического анализатора

Для компиляции программы **lex** выполните следующие действия:

1. С помощью программы **lex** преобразуйте файл спецификаций в программу на языке C. На выходе будет создан файл **lex.yy.c**.
2. С помощью программы **cc** с флагом **-ll** откомпилируйте и скомпонуйте программу с библиотекой **lex**. На выходе будет создан исполняемый файл **a.out**.

Например, если файл спецификаций **lex** называется **lextest**, введите следующие команды:

```
lex lextest
cc lex.yy.c -ll
```

Понятия, связанные с данным:

“Инструменты и утилиты” на стр. 1
В этом разделе приведен обзор инструментов и утилит, предназначенных для разработки программ на языке C.

“Создание синтаксического анализатора с помощью программы yacc” на стр. 511
Программа **yacc** создает синтаксические анализаторы, которые определяют и контролируют структуру текстового ввода компьютерной программы.

Работа с программами lex и yacc

Программа **lex** может также применяться вместе с генератором синтаксических анализаторов, например, с командой **yacc**. Команда **yacc** создает *программу-синтаксический анализатор*, которая может обрабатывать конструкции из нескольких слов.

Такой синтаксический анализатор может работать с лексическим анализатором, создаваемым программой **lex**. Синтаксические анализаторы могут распознавать различные типы грамматик независимо от контекста. Для распознавания лексем синтаксическому анализатору требуется препроцессор - например, анализатор, создаваемый командой **lex**.

Программа **lex** распознает только расширенные регулярные выражения и в соответствии с входным файлом формирует из них пакеты символов, называемые *лексемами*. При использовании программы **lex** для создания препроцессора синтаксического анализатора, лексический анализатор, созданный командой **lex**, применяется для разбиения входного потока. Синтаксический анализатор (созданный командой **yacc**), в

свою очередь, структурирует полученные блоки. С программами, созданными с помощью команд **lex** или **yacc**, могут также использоваться и другие программы.

Лексема - это минимальная независимая смысловая единица, которая может быть определена синтаксическим или лексическим анализатором. Лексема может содержать данные, ключевое слово языка, идентификатор или другой элемент синтаксиса языка.

Программа **yacc** использует функцию лексического анализатора **yylex**, которая присутствует в программах, создаваемых командой **lex**. Обычно функция **yylex** вызывается из главной функции программы **lex**. Однако если установлена команда **yacc** и применяется главная функция этой команды, то **yacc** также вызывает функцию **yylex**. В этом случае при возврате значения лексемы (`token`) каждое правило программы **lex** должно завершаться следующим оператором:

```
return(token);
```

Команда **yacc** присваивает целое значение каждой лексеме, определенной в файле грамматики **yacc** с помощью команды препроцессора C #**define**. У лексического анализатора должен быть доступ к этим макроопределениям, чтобы он мог возвращать лексемы синтаксическому анализатору. Команда **yacc -d** позволяет создать файл **y.tab.h**; этот файл **y.tab.h** затем должен быть подключен к файлу спецификаций **lex** путем добавления следующей строки в раздел определений файла спецификаций **lex**:

```
%{
#include "y.tab.h"
%}
```

Вы можете также включить файл **lex.yy.c** в программу, созданную **yacc**, добавив следующую строку после второго разделителя `%%` (двойной знак процента) в файле грамматики **yacc**:

```
#include "lex.yy.c"
```

Библиотека **yacc** должна загружаться до библиотеки **lex**, поскольку только в этом случае функция main будет вызывать синтаксический анализатор **yacc**. Вы можете создавать программы **lex** и **yacc** в произвольном порядке.

Расширенные регулярные выражения в команде lex

Определение расширенных регулярных выражение в файле спецификаций **lex** похоже на способы, применяемые в командах **sed** и **ed**.

Расширенное регулярное выражение определяет набор строк для сравнения. Выражение содержит как текстовые символы, так и символы операторов. Текстовые символы сравниваются с символами во входной строке. Операторы задают повторы, операции выбора и другие операции.

Цифры и буквы алфавита считаются текстовыми символами. Например, расширенное регулярное выражение `integer` соответствует строке `integer`, а выражение `a57D` - строке `a57D`.

Операторы

Ниже приведено описание операторов, которые могут применяться при определении расширенных регулярных выражений:

Символ
: Соответствует символу *Символ.*

  Пример:`a` соответствует символу a; `b` соответствует символу b, `c` соответствует символу c.

"*Строка***"**
: Соответствует строке, заключенной в кавычки; операторы внутри строки не учитываются.

  Пример: Для того чтобы команда **lex** не интерпретировала символ $ (знак доллара) как оператор, заключите этот символ в кавычки.

символ **или** *цифры*

Escape-символ. Если символ \ расположен перед символьным оператором в строке, это означает, что символ оператора должен интерпретироваться как литерал, а не как оператор. В число допустимых escape-символов входят:

\a Предупреждение

\b Забой

\f Новая страница

\n Символ перехода на новую строку (использовать сам символ новой строки в выражениях нельзя.)

\r Возврат каретки

\t Табуляция

\v Вертикальная табуляция

\\ Обратная косая черта

цифры

Символ, код которого указан одной, двумя или тремя восьмеричными *цифрами*.

\x*цифры*

Символ, код которого указан последовательностью шестнадцатеричных *цифр*.

Если символ \ стоит перед символом, который отсутствует в приведенном выше списке escape-символов, то команда **lex** интерпретирует его как литерал.

Пример: `\c` интерпретируется как символ `c`, а `[\^abc]` обозначает класс символов, включающий символы `^abc`.

Примечание: Не используйте символы `\0` и `\x0` в правилах **lex**.

[*список***]**

Соответствует любому символу из указанного диапазона ([*x*-*y*]) или списка ([*xyz*]), с учетом локали, в которой выполняется команда **lex**. Все операторы, кроме перечисленных ниже, интерпретируются как символы при указании в квадратных скобках: **-** (дефис), ^ (знак вставки) и \ (обратная косая черта).

Пример: `[abc-f]` соответствует символам `a`, `b`, `c`, `d`, `e` и `f` в локали en_US.

[:*класс***:]**

Соответствует любому символу, принадлежащему классу, указанному в ограничителях `[::]`, в соответствии с категорией LC_TYPE текущей локали. Следующие имена классов поддерживаются во всех локалях:

```
alnum   cntrl  lower   space

alpha   digit   print  upper

blank  graph   punct   xdigit
```

Команда **lex** поддерживает также пользовательские классы символов. Оператор `[::]` может применяться только в выражении `[]`.

Пример: `[[:alpha:]]` соответствует любому символу из класса символов **alpha** текущей локали, однако `[:alpha:]` соответствует только символам `:`,`a`,`l`,`p` и `h`.

[.*составной-символ***.]**

Соответствует составному символу, указанному в ограничителях `[..]`, и рассматривается как один символ. Оператор `[..]` может применяться в выражении `[ ]`. Составной символ должен быть допустимым составным символом в текущей локали.

Пример: `[[.ch.]]` соответствует символам `c` и `h` вместе, в то время как `[ch]` соответствует `c` или `h`.

[=*эквивалентный-элемент***=]**

Соответствует элементу, указанному в ограничителях [==] и всем элементам того же класса эквивалентности. Оператор [==] может применяться только в выражении [].

Пример: Если w и v относятся к одному классу эквивалентности, [[=w=]] совпадает с [wv] и соответствует w и v. Если w не принадлежит классу эквивалентности, [[=w=]] соответствует только w.

[^*символ***]**

Соответствует любому символу за исключением символа, следующего за символом ^ (знаком вставки). Результирующий класс символов содержит только однобайтовые символы. После знака ^ можно указать многобайтовый символ. Однако для применения многобайтовых символов в этом операторе значения **%h** и **%m** , заданные в разделе определений, должны быть больше нуля.

Пример: [^c] соответствует любому символу, кроме c.

элемент **-***элемент*

В классе символов определяет диапазон символов в соответствии с последовательностью упорядочения текущей локали. Диапазоны должны указываться в возрастающем порядке. Порядковый номер конечного символа должен быть большим или равным номеру начального. В связи с тем, что диапазон зависит от текущей локали, фиксированный диапазон может соответствовать разным наборам символов, в зависимости от того, в какой локали запущена команда **lex**.

*выражение***?**

Соответствует не более чем одному вхождению выражения, расположенного перед оператором ? .

Пример: ab?c соответствует ac и abc.

Точка (.)

Соответствует любому символу, кроме символа новой строки. Для того чтобы символ точки (.) соответствовал многобайтовым символам, в разделе определений файла спецификаций **lex** значение **%z** должно быть положительным. Если значение **%z** не задано, точка (.) соответствует только однобайтовым символам.

*выражение******

Соответствует нескольким возможным вхождениям выражения, расположенного непосредственно перед оператором *. Например, a* соответствует любому числу символов a, включая ни одного символа. Возможность не указывать ни одного символа полезна в более сложных выражениях.

Пример: выражение [A-Za-z][A-Za-z0-9]* обозначает все алфавитно-цифровые строки, начинающиеся с буквы, включая однобуквенные строки. Это выражение может использоваться для распознавания идентификаторов в языках программирования.

*выражение***+**

Соответствует ненулевому числу вхождений выражения, расположенного непосредственно перед оператором +.

Пример: a+ соответствует одному или нескольким вхождениям aa. [a-z]+ соответствует всем строкам, состоящим из прописных букв.

*выражение***|***выражение*

Обозначает соответствие одному из выражений слева или справа от оператора | (конвейер).

Пример: ab|cd соответствует ab и cd.

(*Выражение***)**

Соответствует выражению в скобках. Оператор () (скобки) применяется для группировки и помещает выражение в скобках в массив **yytext**. Группа, указанная в скобках, может применяться вместо любого отдельного символа в других конструкциях.

Пример: (ab|cd+)?(ef)* соответствует таким строкам, как abefef, efefef, cdef и cddd, но не соответствует строкам abc, abcd и abcdef.

^*выражение*

Соответствует выражению при условии, что *Выражение* находится в начале строки и первым символом в выражении является оператор ^ (знак вставки).

Пример:`^h` соответствует `h` в начале строки.

*выражение***$**

Соответствует выражению при условии, что *Выражение* стоит в конце строки и последним символом в выражении является оператор $ (символ доллара).

Пример: `h$` соответствует `h` в конце строки.

*выражение-1***/***выражение-2*

Соответствует выражению-1, при условии, что *выражение-2* следует непосредственно за *выражением-1*. Оператор / (косая черта) помещает в массив **yytext** только первое выражение.

Пример: `ab/cd` соответствует строке `ab` при условии, что за ней следует строка `cd`; при этом `ab` считывается в массив **yytext**.

Примечание: В одном расширенном регулярном выражении может применяться только один оператор завершающего контекста /. Операторы ^ (знак вставки) и $ (знак доллара) нельзя указывать в том же выражении, что и оператор /, поскольку они обозначают особые случаи завершающего контекста.

{*определенное-имя***}**

Соответствует имени, заданному в разделе определений.

Пример: Если вы определили `D` как класс цифр, то `{D}` соответствует любой цифре.

{*число-1***,***число-2* **}**

Соответствует от *число-1* до *число-2* вхождений набора символов, расположенного непосредственно перед оператором. Допустимо использование выражения {*Число*} и {*Число***,**} для указания точного количества вхождений, равного *Число*.

Пример:`xyz{2,4}` соответствует xyzxyz, xyzxyzxyz и xyzxyzxyzxyz. Отличие от операторов +, * and ? состоит в том, что они относятся только к одному символу, расположенному перед оператором. Для того, чтобы выражение относилось только к одному символу, применяется оператор группировки. Например, выражению `xy(z{2,4})` соответствуют строки xyzz, xyzzz и xyzzzz.

<*начальное-состояние***>**

Соответствующее действие выполняется только в том случае, если лексический анализатор находится в указанном начальном состоянии

Пример: Если начало строки соответствует начальному состоянию `ONE`, то оператор ^ (знак вставки) эквивалентен выражению `<ONE>`.

Для применения символов операторов в качестве текстовых предназначены escape-последовательности `" "` (двойные кавычки) и `\` (обратная косая черта). Оператор `" "` означает, что внутри кавычек расположен обычный текст. Таким образом, строка `xyz++` может быть описана следующим выражением:

```
xyz"++"
```

Можно поместить в кавычки только часть строки. Заключение в кавычки обычного текста не влияет на работу анализатора. Следующее выражение эквивалентно приведенному выше:

```
"xyz++"
```

Для того чтобы гарантировать, что текст будет правильно проинтерпретирован, заключите в кавычки все символы, отличные от букв и цифр.

Символ оператора будет рассматриваться как обычный символ и в том случае, если перед ним будет указан символ `\` (обратная косая черта). Следующее выражение эквивалентно приведенным выше:

```
xyz\+\+
```

Понятия, связанные с данным:

“начальные состояния программы lex” на стр. 510
Правило может быть связано с любым начальным состоянием.

Передача кода в программу, созданную lex

Команда **lex** передает в лексический анализатор неизмененный код на языке C при следующих условиях:

- Строки, начинающиеся с пробела или символа табуляции и расположенные в разделе определений или в начале раздела правил до первого правила, копируются в лексический анализатор без изменений. Если код находится в разделе определений, он копируется в область внешних объявлений файла **lex.yy.c**. Если код находится в разделе правил, он копируется в область локальных объявлений процедуры **yylex** в файле **lex.yy.c**.
- Строки, расположенные между ограничителями %{ (символ процента, левая скобка) и %} (символ процента, правая скобка) в разделе определений или в начале раздела правил, копируются аналогично строкам, начинающимся с пробелов и символов табуляции.
- Все строки, расположенные после второго разделителя %% (два символа процента), копируются в лексический анализатор без ограничения формата.

Определение строк подстановки в lex

Вы можете определить строковые макроопределения, которые команда **lex** должна преобразовывать в строки при создании текста лексического анализатора.

Макроопределения должны быть определены перед первым ограничителем %% в файле спецификаций **lex**. Любая строка в этом разделе, начинающаяся с 1 столбца и не находящаяся внутри блока, ограниченного символами %{ и %}, определяет строку подстановки **lex**. Определения строк подстановки задаются в следующем формате:

```
имя                     значение
```

Поля имя и значение должны быть разделены как минимум одним пробелом или символом табуляции. Имя должно начинаться с буквы. Когда программа **lex** обнаруживает строку, определенную значением имя, заключенным в {} (фигурные скобки) в разделе правил файла спецификаций, она заменяет это имя на строку, заданную в поле значение, и удаляет фигурные скобки.

Например, для определения имен D и E необходимо поместить следующие строки перед первым ограничителем %% в файле спецификаций:

```
D           [0-9]
E           [DEde][-+]{D}+
```

Теперь эти имена можно применять в разделе правил для сокращения текста:

```
{D}+                        printf("целое число");
{D}+"."{D}*({E})?           |
{D}*"."{D}+({E})?           |
{D}+{E}                     printf("действительное число");
```

В раздел определений могут быть также включены следующие объекты:

- Таблица набора символов
- Список начальных состояний
- Изменение размера массивов для поддержки исходных текстов большего объема

Библиотека lex

Библиотека **lex** содержит следующие процедуры:

Функция	Описание
main()	Вызывает лексический анализатор с помощью функции **yylex**.
yywrap()	Возвращает 1 при завершении входных данных.
yymore()	Добавляет следующую найденную строку в конец текущего массива **yytext** вместо замены содержимого массива **yytext**.
yyless(int*n*)	Оставляет *n* первых символов в массиве **yytext** и возвращает остальные символы из массива во входной поток.
yyreject()	Позволяет лексическому анализатору сравнить несколько правил с одной строкой ввода. (Функция **yyreject** вызывается при выполнении специального действия **REJECT**.)

Некоторые функции **lex** могут быть заменены пользовательскими. Например, команда **lex** поддерживает пользовательские версии процедур **main** и **yywrap**. Базовые функции этих процедур, составляющие основу для доработки, приведены ниже:

Функция main

```
#include<stdio.h>
#include <locale.h>
main() {
     setlocale(LC_ALL, "");
     yylex();
     exit(0);
}
```

yywrap subroutine

```
yywrap() {
       return(1);
}
```

Функции **yymore**, **yyless** и **yyreject** доступны только в библиотеке **lex**. Однако они применяются только в действиях команды **lex**.

Действия, выполняемые лексическим анализатором

После того как лексический анализатор найдет соответствие между входной строкой и одним из расширенных регулярных выражений, описанных в разделе правил файла спецификаций, он выполняет *действие*, связанное в этим выражением. Если для обработки входного потока определено недостаточное количество правил, анализатор просто копирует его в выходной поток. Поэтому создавать правила, которые просто копируют текст из входного потока в выходной, не следует. Показываемый по умолчанию вывод позволяет найти ошибки в правилах.

Если команда **lex** применяется для обработки входных данных синтаксического анализатора, создаваемого командой **yacc**, то необходимо задать правила для обработки всех возможных строк. Эти правила должны выводить информацию в формате, понятном команде **yacc**.

Пустое действие

Для того чтобы пропустить текст, связанный с расширенным регулярным выражением, укажите ; (пустой оператор C) в качестве кода действия. Приведенный ниже пример пропускает три символа форматирования (пробел, табуляцию и переход на новую строку):

```
[ \t\n] ;
```

Аналогично следующему действию

Для того чтобы не повторять код действия несколько раз, применяется символ | (символ конвейера). Этот символ означает, что для выражения используется то же действие, что и для следующего выражения. Например, приведенный выше пример, в котором игнорируются пробелы, символы табуляции и символы новой строки, можно записать следующим образом:

```
" "                     |
"\t" |
"\n" ;
```

Символы \n и \t можно не заключать в кавычки.

Вывод найденной строки

Для того чтобы узнать, какой текст был сопоставлен выражению, указанному в разделе правил файла спецификаций, добавьте в список действий для этого выражения вызов функции **printf** языка C. После того как лексический анализатор обнаруживает во входном потоке строку, соответствующую одному из правил, он помещает ее во внешние массивы однобайтовых (**char**) и многобайтовых (**wchar_t**) символов, называемые **yytext** и **yywtext** соответственно. Для вывода обнаруженной строки может, например, использоваться следующий код:

```
[a-z]+       printf("%s",yytext);
```

При вызове функции **printf** необходимо указать формат и данные для вывода. В данном примере аргументы функции **printf** имеют следующее значение:

%s Этот символ указывает на то, что перед выводом данные должны быть преобразованы к строчному формату

%S Этот символ преобразует данные к строчному формату с поддержкой многобайтовых символов (**wchar_t**)

yytext
Имя массива, в котором находятся выводимые данные

yywtext
Имя массива, содержащего многобайтовые данные (**wchar_t**)

Команда **lex** определяет специальное действие **ECHO** для вывода массива **yytext**. Например, следующие два правила эквивалентны друг другу:

```
[a-z]+       ECHO;
[a-z]+       printf("%s",yytext);
```

Представление **yytext** можно изменить, указав значение **%array** или **%pointer** в разделе определений файла спецификаций **lex**:

%array
Определяет **yytext** как массив символов, оканчивающийся нулем. Это действие по умолчанию.

%pointer
Определяет **yytext** как указатель на строку, оканчивающуюся нулем.

Определение длины найденной строки

Для того чтобы определить число символов в строке, соответствующей расширенному регулярному выражению, применяются внешние переменные **yyleng** и **yywleng**.

yyleng
Число байт в найденной строке.

yywleng
Число многобайтовых символов в найденной строке. Размер многобайтовых символов больше 1.

Для подсчета числа слов и числа символов во входном потоке используйте следующее действие:

```
[a-zA-Z]+       {words++;chars += yyleng;}
```

Это действие подсчитывает общее число символов в найденных словах и помещает его в переменную chars.

Следующее выражение возвращает последний символ в найденной строке:

```
yytext[yyleng-1]
```

Сравнение строк

Команда **lex** разбивает входной поток и не обеспечивает поиск всех возможных соответствий каждого выражения. Каждый символ учитывается только один раз. Если необходимо найти строки, которые могут перекрываться и содержать друг друга, используйте действие **REJECT**. Например, для подсчета всех вхождений she и he, включая вхождения he внутри she, опишите следующие действия:

```
she             {s++; REJECT;}
he              {h++}
\n              |
.               ;
```

После подсчета числа вхождений she команда **lex** отклоняет входной поток, а затем подсчитывает число вхождений he. Так как he не может включать she, действие **REJECT** для he указывать необязательно.

Добавление результатов в массив yytext

В обычном режиме следующая строка из входного потока заменяет текущую строку в массиве **yytext**. При вызове процедуры **yymore** следующая строка добавляется в конец текущей строки в массиве **yytext**.

Например, следующий лексический анализатор ищет строки:

```
%s instring
%%
<INITIAL>\" { /* начало строки */
         BEGIN instring;
         yymore();
        }
<instring>\" { /* конец строки */
         printf("найдено %s\n", yytext);
         BEGIN INITIAL;
        }
<instring>.     {
         yymore();
        }
<instring>\n {
         printf("Ошибка, обнаружен символ новой строки \n");
         BEGIN INITIAL;
        }
```

Несмотря на то, что строка может быть обнаружена несколькими правилами, повторяющиеся вызовы **yymore** гарантируют, что в массив **yytext** будет помещена вся строка.

Возврат символов в поток ввода

Для возврата символов в поток ввода применяется следующий вызов:

```
yyless(n)
```

здесь n обозначает сохраняемое число символов текущей строки. Остальные символы будут помещены обратно во входной поток. Функция **yyless** выполняет те же действия, что и оператор / (косая черта), однако позволяет более точно управлять ее работой.

Функция **yyless** используется в случаях, когда текст нужно обработать несколько раз. Например, при анализе программ на языке C встречаются трудные для анализа выражения, такие как x=-a. Это может означать, что переменной x *присваивается значение, противоположное* a, или что это устаревший вызов x -= a,

означающий *уменьшение* значения x *на величину* a." Для того чтобы это выражение рассматривалось как присвоение переменной x *значения, противоположного* a, и при этом выводилось предупреждение, можно задать следующее правило:

```
=-[a-zA-Z]      {
                printf("Оператор (=-) не поддерживается\n");
                yyless(yyleng-1);
                ... действие для = ...
                }
```

Функции ввода-вывода

Команда **lex** позволяет использовать в программе следующие функции ввода-вывода:

input()
Возвращает следующий символ ввода

output(c)
Записывает символ c в поток вывода

unput(c)
Вставляет символ c обратно в поток ввода, для того чтобы его можно было считать функцией **input**

winput()
Возвращает следующий многобайтовый символ ввода

woutput(C)
Записывает многобайтовый символ C в поток вывода

wunput(C)
Вставляет многобайтовый символ C обратно в поток ввода, чтобы его можно было считать функцией **winput**

Программа **lex** предоставляет эти функции как макроопределения. Они хранятся в файле **lex.yy.c**. Вы можете создать другую реализацию этих процедур.

Макроопределения **winput**, **wunput** и **woutput** применяют функции **yywinput**, **yywunput** и **yywoutput**. Из соображений совместимости процедуры **yy** применяют соответствующие процедуры **input**, **unput** и **output** для чтения, замены и записи требуемого числа байт для полного многобайтового символа.

Эти процедуры определяют связь между внешними файлами и внутренними символами. Если вы будете изменять эти процедуры, то изменяйте все процедуры сходным образом. Они должны соответствовать следующим правилам:

- Все процедуры должны использовать одинаковый набор символов.
- При достижении конца файла процедура **input** должна возвращать значение 0.
- Не изменяйте взаимосвязь между процедурой **unput** и процедурой **input**, либо опережающие функции не будут работать.

Файл **lex.yy.c** позволяет лексическому анализатору хранить до 200 символов.

Для работы с файлом, содержащим нулевые значения, необходимо создать новую версию функции **input**. Стандартная версия **input** использует нулевое значение (возвращаемое также нулевыми символами) как индикатор конца файла и окончание ввода.

Набор символов

Лексический анализатор, создаваемый командой **lex**, выполняет ввод-вывод символов с помощью функций **input, output** и **unput**. Таким образом, для возврата значений в процедуре **yytext** команда **lex** использует представление символов, которое применяется в этих процедурах. Однако в качестве внутреннего представления символов команда **lex** использует целые числа. При работе со стандартной библиотекой эти

числа соответствуют наборам битов, которые используются компьютером для обозначения символов. Обычно буква *a* представляется так же, как и символьная константа *a*. Если вы измените такое соответствие в своих процедурах ввода-вывода, то в раздел определений файла спецификаций потребуется добавить таблицу преобразования. Таблица преобразования должна начинаться и заканчиваться строкой, содержащей только следующий текст:

```
%T
```

Внутри таблицы преобразования располагаются строки, устанавливающие соответствие между значениями и символами. Например:

```
%T
{целое число}   {строка символов}
{целое число}   {строка символов}
{целое число}   {строка символов}
%T
```

Конец файла, обработка

Когда лексический анализатор доходит до конца файла, он вызывает библиотечную функцию **yywrap**, которая возвращает значение 1, чтобы указать лексическому анализатору, что он должен завершить обработку потока ввода.

Однако в тех случаях, когда лексический анализатор может получать данные из нескольких источников, необходимо изменить процедуру **yywrap**. Новая функция должна сформировать новый поток ввода и вернуть нулевое значение. Значение 0 указывает, что программа должна продолжить обработку.

В новую версию функции **yywrap** может быть также добавлен код, который показывает итоговые таблицы и суммарные значения. Изменение процедуры **yywrap** - это единственный способ указать процедуре **yylex** на продолжение или завершение входного потока.

начальные состояния программы lex

Правило может быть связано с любым начальным состоянием.

Однако правило будет учитываться программой **lex** только тогда, когда она находится в указанном состоянии. Текущее начальное состояние можно изменить в любой момент.

Начальные состояния указываются в разделе *определений* файла спецификаций с помощью следующей конструкции:

```
%Start  имя-1 имя-2
```

Здесь `имя-1` и `имя-2` определяют имена, соответствующие состоянию. Число состояний не ограничено, их порядок значения не имеет. Слово `Start` можно сократить до `s` или `S`.

При описании правила, соответствующего начальному состоянию, имя состояния в начале правила заключается в символы <> (угловые скобки). В приведенном ниже примере определено правило `expression`, которое распознается программой **lex** только в том случае, если она находится в начальном состоянии `name1`:"

```
<name1> expression
```

Для перевода программы **lex** в требуемое состояние применяется действие `BEGIN`:

```
BEGIN name1;
```

Это действие устанавливает состояние `name1`.

Для перехода в обычное состояние введите:

```
BEGIN 0;
```

или

```
BEGIN INITIAL;
```

В этом примере INITIAL соответствует значению 0 согласно определениям программы **lex**. BEGIN 0; возвращает **lex** в исходное состояние.

Программа **lex** поддерживает также исключительные состояния запуска. Они определяются с помощью операторов %**x** (знак процента, строчная x) и %**X** (знак процента, заглавная X), за которым следует список имен исключительных состояний, аналогично списку обычных состояний запуска. Исключительные состояния отличаются от обычных тем, что в исключительном состоянии строки, для которых состояние не указано, не учитываются анализатором. Например:

```
%s      one
%x      two
%%
abc {printf("matched ");ECHO;BEGIN one;}
<one>def printf("matched ");ECHO;BEGIN two;}
<two>ghi {printf("matched ");ECHO;BEGIN INITIAL;}
```

В состоянии one будут распознаваться и строка abc, и строка def. В состоянии two будет распознаваться только строка ghi.

Понятия, связанные с данным:

“Расширенные регулярные выражения в команде lex” на стр. 501
Определение расширенных регулярных выражение в файле спецификаций **lex** похоже на способы, применяемые в командах **sed** и **ed**.

Создание синтаксического анализатора с помощью программы yacc

Программа **yacc** создает синтаксические анализаторы, которые определяют и контролируют структуру текстового ввода компьютерной программы.

Для работы с этой программой необходимо определить следующий элементы:

файл грамматики
Исходный файл, содержащий спецификации распознаваемого языка. В этом файле также находятся определения функций **main**, **yyerror** и **yylex**. Это обязательные функции.

main
Функция языка C, которая, как минимум, вызывает функцию **yyparse**, созданную программой **yacc**. Ограниченная версия этой функции присутствует в библиотеке **yacc**.

yyerror
Функция языка C, предназначенная для обработки ошибок, возникающих при работе синтаксического анализатора. Ограниченная версия этой функции присутствует в библиотеке **yacc**.

yylex
Функция языка C, выполняющая лексический анализ входного потока и передающая лексемы синтаксическому анализатору. Лексический анализатор можно создать с помощью команды **lex**.

На основе файла спецификаций программа **yacc** создает файл **y.tab.c** на языке C. После компиляции файла командой **cc** формируется код функции **yyparse**, возвращающей целое число. При работе функция **yyparse** вызывает для чтения лексем функцию **yylex**. Функция **yylex** возвращает лексемы до тех пор, пока синтаксический анализатор не обнаружит ошибку или **yylex** не вернет маркер конца, означающий завершение входного потока. При возникновении неустранимой ошибки функция **yyparse** возвращает в функцию **main** значение 1. При обнаружении маркера конца функция **yyparse** возвращает в функцию **main** значение 0.

Понятия, связанные с данным:

“Создание лексического анализатора с помощью команды lex” на стр. 499
Команда **lex** помогает создать программу на языке C, которая может получать поток символов и преобразовывать его в действия программы.

Файл грамматики yacc

Для создания синтаксического анализатора необходимо передать на вход команды **yacc** файл грамматики, который описывает формат входного потока и определяет действия, которые анализатор будет выполнять над данными.

Файл грамматики включает правила, описывающие структуру ввода, код, который будет вызываться при распознавании таких правил, и функцию, отвечающую за ввод.

На основании информации из файла грамматики команда **yacc** создает синтаксический анализатор, управляющий процессом ввода. Для получения элементов входного потока (*лексем*) этот анализатор вызывает процедуру ввода (лексический анализатор). Лексема - это символ или имя, сообщающее синтаксическому анализатору о том, какой символьный шаблон передается ему функцией ввода. Нетерминальный символ определяет структуру, распознаваемую анализатором. Затем он структурирует лексемы в соответствии с правилами, заданными в файле грамматики. Правила структурирования лексем называются *грамматическими правилами*. Когда анализатор распознает одно из заданных правил, он выполняет связанный с этим правилом пользовательский код. Этот пользовательский код называется *действием*. Действия возвращают значения и используют значения, возвращаемые другими действиями.

Для создания кода действий и других функций применяется язык программирования C. Многие синтаксические конструкции, используемые в файле грамматики **yacc**, также заимствованы из языка C.

Функции main и yyerror

Для работы синтаксического анализатора должны быть определены функции **main** и **yyerror**. Для упрощения работы с **yacc** в библиотеку **yacc** включены краткие варианты функций **main** и **yyerror**. Для подключения этих определений функций применяется аргумент *-ly* команды **ld** (или команды **cc**). Ниже приведен исходный код библиотечной версии **main**:

```
#include <locale.h>
main()
{
     setlocale(LC_ALL, "");
     yyparse();
}
```

Библиотечная версия функции **yyerror** определена следующим образом:

```
#include <stdio.h>
yyerror(s)
        char *s;
{
        fprintf( stderr, "%s\n" ,s);
}
```

Аргументом **yyerror** является строка, содержащая сообщение об ошибке - обычно это строка `syntax error`.

Эти программы можно дополнить необходимыми функциями. Например, вы можете отслеживать номер строки входного текста и указывать его в сообщениях об ошибках. Вы можете также использовать значение внешней переменной *yychar*. На момент обнаружения ошибки в этой переменной хранится номер следующей лексемы.

Функция yylex

Функция ввода, предоставляемая для файла грамматики, должна выполнять следующие операции:

- Считывать поток ввода.

- Распознавать основные шаблоны в потоке ввода.
- Передавать эти шаблоны и соответствующие им лексемы анализатору.

Допустим, что функция ввода разбивает поток ввода на лексемы `WORD`, `NUMBER` и `PUNCTUATION`. Ей был передан следующий ввод:

```
I have 9 turkeys.
```

В этом случае функция ввода передаст анализатору следующие строки и лексемы:

Строка	Маркер
`I`	WORD
`have`	WORD
`9`	NUMBER
`turkeys`	WORD
`.`	PUNCTUATION

Синтаксический анализатор должен включать определения лексем, передаваемых ему функцией ввода. Опция **-d** команды **yacc** позволяет создать список лексем в файле **y.tab.h**. Этот список представляет собой набор операторов **#define** и позволяет использовать одни и те же лексемы в лексическом (**yylex**) и синтаксическом анализаторах.

Примечание: Для того чтобы избежать конфликтов с синтаксическим анализатором, не используйте имена, начинающиеся с букв `yy`.

Функцию ввода можно создать с помощью команды **lex**, либо написать ее самостоятельно на языке C.

Применение файла грамматики yacc

Файл грамматики **yacc** содержит следующие разделы:

- Объявления
- Правила
- Программы

Разделы файла грамматики отделяются друг от друга символами %% (двойной знак процента). Обычно символы %% размещают на отдельных строках, что упрощает чтение файла. Общая структура файла грамматики выглядит следующим образом:

```
объявления
%%
правила
%%
программы
```

Раздел объявлений может быть пустым. Если в файле отсутствует раздел программ, не вводите вторую пару %%. Таким образом, минимальный файл грамматики **yacc** выглядит следующим образом:

```
%%
правила
```

Команда **yacc** игнорирует пробелы, символы табуляции и символы новой строки, содержащиеся в файле грамматики. Таким образом, эти символы могут применяться для форматирования файла грамматики. Однако пробелы, символы табуляции и символы новой строки нельзя использовать в именах и зарезервированных словах.

Комментарии

Для объяснения работы анализатора в файл грамматики можно добавить комментарии. Комментарий можно разместить в любом месте файла грамматики, в котором можно указать имя. Однако для улучшения

читаемости файла рекомендуется помещать комментарии на отдельных строках в начале смысловых блоков или правил. Комментарий в файле грамматики **yacc** обозначается так же, как и в тексте программы на C. Комментарий заключается в символы /* (косая черта, звездочка) и */ (звездочка, косая черта). Пример:

```
/* Это отдельная строка комментария. */
```

Литеральные строки

Литеральная строка представляет собой один или несколько символов, заключенных в (одиночные кавычки) ' '. Как и в языке C, символ \ (обратная косая черта) определяет escape-символ внутри литерала. Распознаются все escape-коды языка C. Таким образом, команда **yacc** поддерживает символы из следующей таблицы:

Символ	Определение
'**a**'	Предупреждение
'**b**'	Забой
'**f**'	Новая страница
'**n**'	Символ новой строки
'**r**'	Возврат каретки
'**t**'	Табуляция
'**v**'	Вертикальная табуляция
'\''	Одиночная кавычка (')
'\"'	Двойная кавычка (")
'\?'	Вопросительный знак (?)
'\\'	Обратная косая черта (\)
'*цифры*'	Символ, код которого указан одной, двумя или тремя восьмеричными *цифрами*.
'**x***цифры*'	Символ, код которого указан последовательностью шестнадцатеричных *цифр*.

Поскольку код ASCII символа NULL (\0 или 0) равен нулю, его не следует использовать в файле грамматики. При обнаружении символа NULL функция **yylex** возвращает 0, что означает конец ввода.

Форматирование файла грамматики

Для улучшения читаемости файла грамматики **yacc** следуйте приведенным ниже рекомендациям:

- Указывайте имена лексем прописными, а имена нетерминальных символов - строчными буквами.
- Указывайте грамматические правила и действия на отдельных строках - это позволит изменить один элемент, не меняя другого.
- Группируйте вместе правила с одинаковой левой частью. Введите левую часть один раз и используйте вертикальную черту для задания остальных правил с такой левой частью.
- В конце каждого набора правил с одинаковой левой частью введите на отдельной строке одну точку с запятой. Это позволит легко добавлять новые правила.
- Отделяйте текст правил двумя символами табуляции, а код действий - тремя.

Ошибки в файле грамматики

Команда **yacc** не может создать анализатор для всех возможных наборов грамматик. Если правила внутренне противоречивы или используют методы сравнения, недоступные в **yacc**, то команда **yacc** не сможет создать анализатор. В большинстве случаев при возникновении ошибки программа **yacc** выводит сообщение. Для исправления ошибок исправьте правила в файле грамматики или создайте лексический анализатор (программу ввода для синтаксического анализатора), который распознает последовательности, не поддерживаемые **yacc**.

Объявления в файле грамматики yacc

Раздел объявлений файла грамматики **yacc** содержит следующую информацию:

- Объявления переменных и констант, применяемых в других разделах файла грамматики

- Директивы #**include** для включения других файлов в файл грамматики (применяются для подключения заголовочных файлов библиотек)
- Операторы, определяющие условия работы созданного синтаксического анализатора

Семантическая информация, связанная с лексемами, находящимися в стеке анализа, может храниться в определенном пользователем объединении (*union*) языка C, если элементы этого объединения связаны с различными именами из файла грамматики.

Объявления переменных и констант задаются в соответствии с синтаксическими правилами языка C:

СпецификаторТипа Идентификатор **;**

СпецификаторТипа - это ключевое слово, определяющее тип данных, *Идентификатор* - имя переменной или константы. Имена могут иметь произвольную длину и содержать буквы, символы подчеркивания и цифры. Имя не может начинаться с цифры. В именах учитывается регистр символов.

Имена терминальных символов (лексем) описываются с помощью объявления **%token**, а имена нетерминальных символов - с помощью объявления **%type**. Объявлять нетерминальные символы с помощью **%type** необязательно. Они определяются автоматически, когда программа обнаруживает эти символы в левой части какого-либо правила. Если имя не определено в разделе объявлений, оно может применяться только в качестве нетерминального символа. Синтаксис и функции операторов **#include** аналогичны синтаксису и функциям одноименных директив языка C.

В программе **yacc** определен набор ключевых слов, которые применяются для описания условий обработки. Каждое ключевое слово начинается с символа **%** (знак процента), за которым следует лексема или нетерминальный символ. Предусмотрены следующие ключевые слова:

Ключевое слово	Описание
%left	Определяет лексемы с ассоциативностью слева.
%nonassoc	Определяет лексемы без ассоциативности.
%right	Определяет лексемы с ассоциативностью справа.
%start	Обозначает нетерминальное имя символа начала.
%token	Определяет имена лексем, распознаваемые **yacc**. Позволяет описать все имена лексем в разделе объявлений.
%type	Задает тип нетерминальных символов. Если это объявление присутствует, выполняется проверка типов.
%union	Описывает стек значений yacc в виде объединения с элементами требуемых типов. По умолчанию возвращаются целые числа. Эта конструкция позволяет объявить **YYSTYPE** непосредственно на основе входных данных.
%{ Код %}	Копирует *Код* в исходный файл программы. Это ключевое слово позволяет добавлять в раздел объявлений описания и определения на языке C. **Примечание:** Символы %{ (процент, левая фигурная скобка) и %} (процент, правая фигурная скобка) должна располагаться на отдельных строках.

Ключевые слова **%token**, **%left**, **%right** и **%nonassoc** могут поддерживать имя элемента объединения C (определенное с помощью **%union**), обозначаемое как **<***Tag***>** (имя элемента объединения в угловых скобках). В ключевом слове **%type** должен быть указан **<***Тег***>**. Элемент **<***Тег***>** позволяет указать, что перечисленные лексемы должны быть того же типа, что и элемент объединения C, на который ссылается **<***Тег***>**. Например, в следующей строке параметр *Имя* объявляется в качестве лексемы:

```
%token [<Тег>] Имя [Число] [Имя [Число]]...
```

Если <*Тег*> указан, то все лексемы, перечисленные в этой строке, должны иметь тот же тип (языка C), что и <*Тег*>. Если в объявлении присутствует положительное целое *Число* после параметра *Имя*, то это число присваивается лексеме.

Все лексемы, перечисленные в строке, имеют одинаковый приоритет и ассоциативность. Строки файла расположены в порядке возрастания приоритета. Например, следующие строки описывают приоритет и ассоциативность четырех арифметических действий:

```
%left '+' '-'
%left '*' '/'
```

Символы **+** (плюс) и **-** (минус) имеют ассоциативность слева и меньший приоритет, чем символы ***** (звездочка) и **/** (косая черта), которые также имеют ассоциативность слева.

Определение глобальных переменных

При объявлении переменных, которые будут использоваться в действиях и в лексическом анализаторе, заключайте их определения в символы %{ (процент, левая фигурная скобка) и %} (процент, правая фигурная скобка). Объявления, заключенные в эти символы, называются *глобальными переменными*. Например, для того чтобы переменная **var** была доступна во всех частях программы, добавьте в раздел объявлений файла грамматики следующие строки:

```
%{
int var = 0;
%}
```

Начальные состояния

Анализатор распознает специальный символа, называемый *начальным*. Начальный символ - это имя правила, определенного в разделе правил файла грамматики, которое описывает наиболее общую структуру анализируемого языка. Так как это самая общая структура, анализатор начинает грамматический разбор входного потока именно с нее. Для описания начального символа в разделе объявлений применяется ключевое слово **%start**. Если начальный символ не определен, анализатор использует первое правило из раздела правил файла грамматики.

Например, при анализе функции языка C наиболее общей является следующая структура:

```
main()
{
        code_segment
}
```

Начальный символ должен указывать на правило, описывающее такую структуру. Все остальные правила файла будут описывать структуры более низкого уровня, используемые внутри функции.

Номера лексем

Номера лексем - это неотрицательные целые числа, соответствующие именам лексем. Если лексический анализатор передает синтаксическому анализатору номер лексемы вместо ее имени, то обе программы должны использовать одинаковые обозначения лексем.

Номера лексем могут быть заданы в файле грамматики **yacc**. Если номера не заданы явно, то файл грамматики **yacc** назначает номера по следующим правилам:

- Литеральному символу соответствует его код в кодировке ASCII.
- Остальным символам присваиваются коды, начиная с 257.

Примечание: Не присваивайте лексемам нулевое значение. Это значение связано с маркером конца ввода. Переопределять его нельзя.

Для того чтобы присвоить номера лексемам (включая литералы) в разделе объявлений файла грамматики, укажите номер после имени лексемы в строке `%token`. Это число будет номером, соответствующим имени лексемы или литерала. Номера лексем должны быть уникальными. По достижении конца ввода лексический анализатор, используемый совместно с **yacc**, должен возвращать 0 или отрицательное значение.

Правила yacc

Раздел правил файла грамматики содержит одно или несколько грамматических правил. Каждое правило описывает структуру и присваивает ей имя.

Грамматические правила задаются в следующем виде:

```
A : ТЕКСТ;
```

Здесь `A` - нетерминальное имя, а `ТЕКСТ` - последовательность из 0 или более имен, литералов и семантических действий, после которых могут следовать правила приоритета. Для описания грамматики необходимы только имена и литералы. Семантические действия и правила приоритетов необязательны. Двоеточие и точки запятой должны быть указаны в определениях правил **yacc** обязательно.

Семантические действия позволяют выполнять определенный код при каждом распознавании правила во входном потоке. В качестве действия может применяться любой оператор C, который будет, например, выполнять ввод-вывод, вызывать функции или изменять значения переменных. Действия могут также вызывать операции анализатора, такие как сдвиг или понижение.

Правила приоритетов определяются ключевым словом **%prec** и изменяют приоритет соответствующего правила грамматики. Зарезервированное слово **%prec** может располагаться непосредственно после текста грамматического правила, в нем может указываться имя лексемы или литерал. При использовании такой конструкции приоритет правила становится равным приоритету имени лексемы или литерала.

Повторение нетерминальных имен

Если нетерминальное имя применяется в нескольких грамматических правилах, то эти правила можно объединить в одно с помощью символа | (символ конвейера). В этом случае символ **;** (точка с запятой) ставится в конце набора объединенных правил. Например, следующие грамматические правила:

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

могут быть описаны в программе **yacc** следующим образом:

```
A  :  B  C  D
   |  E  F
   |  G
   ;
```

Использование рекурсии в файле грамматики

Рекурсия позволяет определять функцию через себя саму. В определении языков такие правила обычно имеют следующую форму:

```
правило    :        КонечноеУсловие
           | rule EndCase
```

Таким образом, в простейшем варианте `правило` соответствует значению *КонечноеУсловие*, но при этом `правило` может содержать произвольное число повторений значения *КонечноеУсловие*. Вторая строка,

использующая правило внутри описания конструкции правило, применяет рекурсию. Анализатор последовательно обрабатывает входной поток, пока он не сократится до единственного значения *КонечноеУсловие*.

Если в правиле используется рекурсия, то имя правила всегда должно быть самым левым в теле правила (как в предыдущем примере). Если оно не будет первым, как в следующем примере, то может возникнуть переполнение стека, в результате чего анализатор прервет работу.

```
правило    :        КонечноеУсловие
        | EndCase rule
```

Приведенный ниже пример определяет правило line (строка) как произвольную комбинацию элементов string (текст), завершающуюся символом новой строки (\n):

```
lines   :        line
        |        lines line
        ;

line    :        string '\n'
        ;
```

Пустая строка

Для обозначения нетерминального символа, соответствующего пустой строке, используйте в качестве тела правила одиночный символ **;** (точка с запятой). Для определения правила empty, соответствующего пустой строке, используйте правило, аналогичное следующему:

```
empty   :  ;
        | x;
```

или

```
empty   :
        | x
        ;
```

Маркер конца ввода

Когда лексический анализатор достигает конца входного потока, он передает синтаксическому анализатору маркер конца ввода. Этот маркер является специальной лексемой со значением 0, называемой *маркером конца*. Когда синтаксический анализатор получает маркер конца, он проверяет, для всех ли входных данных были выбраны правила грамматики и образует ли обработанная информация законченный блок (в соответствии с правилами файла грамматики **уасс**). Если образован законченный блок, анализатор завершает работу. Если блок не образован, анализатор передает сообщение об ошибке и также завершает работу.

Лексический анализатор должен передавать маркер конца в некоторый обоснованный момент - например, по достижении конца файла или конца записи.

Действия уасс

В каждом грамматическом правиле может быть указано действие, которое должно выполняться каждый раз, когда анализатор распознает правило во водном потоке. Действие представляет собой оператор языка C, который может выполнять операции ввода-вывода, вызывать функции, а также изменять значения массивов и переменных.

Действия возвращают и используют значения, возвращаемые другими действиями. Лексический анализатор может также возвращать значения лексем.

Действия задаются в файле грамматики с помощью одного или нескольких операторов, заключенных в фигурные скобки {}. Ниже приведены примеры правил, для которых определены действия:

```
A  :  '('B')'
   {
     hello(1, "abc" );
   }
```

И

```
XXX  :  YYY  ZZZ
     {
     printf("a message\n");
     flag = 25;
     }
```

Передача значений между действиями

Для того чтобы в действии могли использоваться значения, полученные другими действиями, программа может использовать ключевые слова **yacc** для передачи параметров, начинающиеся с символа доллара (`$1`, `$2, ...`). Такие ключевые слова указывают на значения, возвращаемые компонентами в правой части правила, слева направо. Например, для правила

```
A  :  B  C  D  ;
```

параметр `$1` соответствует значению, которое вернуло правило, распознавшее B, `$2` - значению правила, распознавшего C, а `$3` - значению правила, распознавшего D.

Для того чтобы вернуть значение, действие присваивает его псевдопеременной `$$`. Например, следующее действие возвращает значение `1`:

```
{ $$ = 1;}
```

По умолчанию значением правила является значение его первого элемента (`$1`). Таким образом, необязательно указывать действия для правил, имеющих следующий вид:

```
A : B ;
```

Кроме этого, существует набор ключевых слов **yacc**, начинающихся с символа $ (доллар), предназначенных для проверки типов:

- `$<`*Тег*`>$`
- `$<`*Тег*`>`*Число*

`$<`*Тег*`>`*Число* устанавливает для ссылки тип данных элемента объединения, обозначенного как `<`*Тег*`>`. Использование такой конструкции добавляет в ссылку ***.тег*** для работы с элементом объединения, на который указывает *Тег*. Эта конструкция эквивалентна обращению `$$.`*Тег* или `$1.`*Тег*. Эта форма может применяться в действиях, которые располагаются в середине правила, когда тип невозможно определить с помощью объявления **%type**. Если для нетерминального имени применялось объявление **%type**, не используйте конструкцию `<`*Tag*`>` - ссылка на объединение будет создана автоматически.

Размещение действий в середине правил

Для управлением процессом анализа до завершения разбора правила следует размещать действие в середине правила. Если правило возвращает значения с помощью ключевых слов `$`, то действия, следующие за правилом, могут использовать это значение. Такое правило может также использовать значения из предыдущих действий. Оно присваивает `x` значение `1`, а `y` - значение, возвращаемое `C`. Значением правила `A` будет значение, которое вернуло правило `B`, в соответствии с правилом присвоения значений по умолчанию.

```
A  :  B
        {
           $$ =1;
        }
        C
   {
```

```
        x = $2;
        y = $3;
   }
   ;
```

Для такого правила команда **yacc** создает новый внутренний нетерминальный символ, который соответствует действию, расположенному внутри правила. Кроме того, она описывает для этого символа правило, соответствующее пустой строке. Таким образом, приведенный выше пример внутренне представлен в команде **yacc** следующим образом:

```
$ACT  :  /* empty */
      {
         $$ = 1;
      }
      ;
A     :  B  $ACT  C
      {
         x = $2;
         y = $3;
      }
      ;
```

здесь $ACT является пустым действием.

Обработка ошибок в yacc

При чтении анализатором входного потока данные из этого потока могут не соответствовать ни одному правилу файла грамматики.

Анализатор должен обнаруживать подобные ошибки как можно раньше. Если в файле грамматики присутствует функция обработки ошибок, то с ее помощью можно запросить повторный ввод данных, пропустить данные с ошибкой или выполнить очистку и восстановление. Например, при обнаружении ошибки может потребоваться освобождение памяти, выделенной для дерева разбора, удаление или изменение записей таблицы символов и установка флагов для предотвращения дальнейшего разбора.

Если функция обработки ошибок не определена, то при обнаружении ошибки анализатор прекращает работу. Для дальнейшей обработки входного потока с целью обнаружения других ошибок необходимо перезапустить анализатор с той точки, в которой он сможет продолжить разбор входного потока. Одним из способов такого перезапуска является пропуск определенного числа лексем, следующих за лексемой, вызывавшей ошибку.

В команде **yacc** для обработки ошибок применяется специальная лексема **error**. Эту лексему помещают в раздел правил в тех местах, где может возникнуть ошибка и требуются специальные действия по ее обработке. Если ошибка возникает в таком месте, анализатор выполняет вместо обычного действия действие лексемы **error**.

При обработке ошибок в действиях **yacc** могут применяться следующие макроопределения:

Макрокоманды	Описание
YYERROR	Вызывает обработку ошибки анализатором.
YYABORT	Анализатор возвращает значение 1.
YYACCEPT	Анализатор возвращает значение 0.
YYRECOVERING()	Возвращает 1, если была обнаружена синтаксическая ошибка и анализатор еще не завершил процесс исправления.

Для того чтобы одна ошибка не приводила к возникновению нескольких сообщений об ошибках, анализатор остается в состоянии ошибки до те пор, пока он не обработает три лексемы, следующие за лексемой, вызвавшей ошибку. Если в течение этого времени возникнет еще одна ошибка, анализатор пропустит соответствующую лексему без выдачи сообщения.

Например, правило:

```
stat  :  error ';'
```

задает режим обработки, при котором анализатор будет пропускать лексему с ошибкой и все последующие лексемы до символа точки с запятой. Все лексемы, начиная с лексемы, в которой встретилась ошибка, и до точки с запятой, пропускаются. После обнаружения точки с запятой анализатор редуцирует это правило и выполняет установленные действия по очистке.

Исправление ошибок

При возникновении ошибки вы можете предоставить пользователю, работающему с программой в диалоговом режиме, возможность исправить входной текст, запросив повторный ввод строки. Ниже приведен пример такого кода:

```
input : error '\n'
        {
          printf(" Введите последнюю строку еще раз: " );
         }
         input
       {
         $$ = $4;
       }
       ;
```

Однако в этом варианте анализатор остается в состоянии ошибки на протяжении еще трех лексем. Если в одной из 3 первых лексем исправленной строки присутствует ошибка, анализатор пропустит их, не выдав сообщение об ошибке. Для того чтобы учесть подобную ситуацию, используйте следующий оператор **yacc**:

```
yyerrok;
```

Этот оператор переводит анализатор в обычное состояние. Процедура исправления ошибок с этим оператором выглядит так:

```
input : error '\n'
        {
          yyerrok;
          printf(" Введите последнюю строку еще раз: " );
         }
         input
       {
         $$ = $4;
       }
         ;
```

Очистка следующей лексемы

Под *следующей лексемой* подразумевается лексема, которая будет проанализирована программой на следующем шаге. При возникновении ошибки следующая лексема - это лексема, в которой анализатор обнаружил ошибку. Если действие по восстановлению после ошибки определяет точку, с которой обработка будет возобновлена, его код должен очищать следующую лексему. Для очистки значения следующей лексемы применяется следующий оператор:

```
yyclearin ;
```

Операции лексического анализатора для команды yacc

Команда **yacc** преобразует файл грамматики в программу на языке C.

После компиляции и запуска эта программа может обрабатывать входной поток в соответствии с заданной грамматикой.

Синтаксический анализатор представляет собой конечный стековый автомат. Анализатор способен считывать из потока и запоминать следующую лексему. Текущим состоянием всегда является состояние в вершине стека. Состояния конечного автомата определяются целыми числами. Начальное состояние автомата - 0, стек содержит только 0, а следующая лексема не считана.

Автомат может выполнять следующие действия:

Действие	**Описание**
shift*Состояние*	Сдвиг. Анализатор помещает текущее состояние в стек, делает указанное *Состояние* текущим и очищает следующую лексему.
reduce*Правило*	Сокращение. Если анализатор обнаруживает во входном потоке строку, определенную указанным *Правилом* (номером правила), то он заменяет ее в выходном потоке на *Правило*.
accept	Прием. Анализатор просматривает весь входной поток, сравнивает его с определением грамматики и распознает поток, как удовлетворяющий структуре высшего уровня (определяемой начальным символом). Это действие выполняется только в том случае, если следующая лексема - маркер конца, и означает, что анализатор успешно завершил работу.
ошибка	Ошибка. Анализатор не может продолжать обработку входного потока и выборку определенных грамматических правил. Считанные лексемы, совместно со следующей лексемой, ни при каких следующих лексемах не могут сформировать набор, удовлетворяющий одному из правил. Анализатор сообщает об ошибке, пытается исправить ее и продолжить работу.

За один шаг анализатор выполняет следующие действия:

1. В зависимости от текущего состояния, анализатор определяет, требуется ли ему для работы следующая лексема. Если следующая лексема нужна для работы, но она отсутствует, анализатор вызывает функцию **yylex** для считывания очередной лексемы.
2. Используя текущее состояние и, если требуется, следующую лексему, анализатор принимает решение о следующем действии и выполняет его. В результате этого состояние может быть помещено в стек или считано из стека, а следующая лексема может быть обработана или не обработана.

Действие shift

Действие **shift** является наиболее распространенным действием анализатора. Сдвиг выполняется при условии, что существует следующая лексема. Например, рассмотрим следующее грамматическое правило:

```
IF shift 34
```

Если анализатор находится в состоянии, содержащем это правило и следующая лексема - IF, то он выполняет следующие действия:

1. Оставляет текущее состояние в стеке.
2. Делает состояние 34 текущим (помещает его в вершину стека).
3. Очищает следующую лексему.

Действие reduce

Действие **reduce** сокращает стек, что позволяет избежать переполнения. Анализатор применяет это действие после успешного сравнения элемента входного потока с правой частью правила. После этого символы входного потока заменяются на значение из левой части правила. Для определения законченности конструкции анализатор может применять следующую лексему.

Сокращение относится к отдельным грамматическим правилам. В связи с тем, что правилам также присваиваются целые числа, можно легко спутать значения аргументов в действиях **shift** и **reduce**. Например, следующее действие относится к правилу грамматики 18:

```
. reduce 18
```

А это действие относится к состоянию 34:

```
IF shift 34
```

Например, для сокращения следующего правила анализатор считывает из стека три состояния:

```
A  :  x y z ;
```

Число считываемых из стека состояний совпадает с числом символов в правой части правила. Эти состояния были помещены в стек при распознавании x, y и z. После считывания этих состояний в вершине стека оказывается состояние, в котором анализатор находится перед обработкой этого правила; то есть текущим становится состояние, которое требовало распознавания правила A для удовлетворения своего правила. Используя это состояние и символ в левой части правила, анализатор выполняет действие **goto**, которое похоже на сдвиг A. Вычисляется новое состояние, оно помещается в стек и обработка продолжается.

Действие **goto** отличается от обычного сдвига лексемы. Следующая лексема очищается при сдвиге, но не изменяется действием **goto**. После считывания этих трех состояний новое состояние может соответствует, например, такой записи:

```
A  goto 20
```

Эта запись помещает запись состояния 20 в стек (оно становится текущим состоянием).

Действие сокращения также важно при обработке пользовательских действий и значений. При сокращении правила анализатор перед изменением стека выполняет связанное с правилом действие. Еще один стек, существующий одновременно со стеком состояний, хранит значения, которые были возвращены лексическим анализатором и действиями. При сдвиге значение внешней переменной *yylval* копируется в этот стек. После выполнения указанного кода анализатор выполняет сокращение. При выполнении действия **goto** значение внешней переменной *yylval* также копируется в стек значений. Ключевые слова **yacc**, начинающиеся с символа $, работают со стеком значений.

Использование неоднозначных правил в программе yacc

Набор грамматических правил является *неоднозначным*, если какая-либо строка ввода может быть интерпретирована разными способами.

Например, следующее правило определяет выражение, состоящее из двух выражений и символа минус между ними.

```
expr : expr '-' expr
```

Однако такое правило не дает информации о разборе сложных выражений. Например, текст

```
expr - expr - expr
```

может быть разобран с ассоциативностью слева

```
( expr - expr ) - expr
```

или с ассоциативностью справа

```
expr - ( expr - expr )
```

с получением разных результатов.

Конфликты анализатора

При попытке обработать неоднозначное правило в одном из действий анализатора может возникнуть конфликт. Существуют следующие основные типы конфликтов:

Конфликт	Описание
конфликт shift/reduce	Правило может быть правильно обработано действиями **shift** и **reduce**, но результаты, полученные при выполнении этих действий, будут различаться.
конфликт reduce/reduce	Правило может быть правильно обработано двумя способами сокращения, с получением двух разных действий.

Возникновение конфликта **shift/shift** невозможно. Конфликты **shift/reduce** и **reduce/reduce** возникают из-за неполного описания правила. Например, если определено приведенное выше неоднозначное правило, то при получении строки

```
expr - expr - expr
```

анализатор читает первые три части:

```
expr - expr
```

- они соответствуют правой части правила. Поэтому анализатор может сократить их, используя это правило. После сокращения входной поток преобразуется к виду

```
expr
```

- то есть к левой части правила. После этого анализатор считывает оставшуюся часть

```
- expr
```

и сокращает ее. При такой интерпретации реализуется ассоциативность слева.

Однако у анализатора есть возможность упреждающего чтения входного потока. Если после считывания выражения

```
expr - expr
```

анализатор продолжит чтение, он может получить следующее выражение:

```
expr - expr - expr
```

Применение правила к трем правым элементам сократит их до `expr`. После этого будет обработано выражение

```
expr - expr
```

После очередного сокращение будет реализована обработка с ассоциативностью справа.

Таким образом, после считывания первых трех элементов анализатор может выбрать одно из двух допустимых действий - **shift** или **reduce**. Если правило выбора не определено, возникает конфликт **shift/reduce**.

Аналогичная ситуация (называемая конфликтом *reduce/reduce*) возникает, когда анализатор может выбрать одно из двух допустимых действий сокращения.

Реакция анализатора на конфликт

При возникновении конфликта **shift/reduce** или **reduce/reduce** команда **yacc** создает анализатор, выбирая один из допустимых вариантов. Если правило выбора не определено, **yacc** использует следующие правила:

- При конфликтах **shift/reduce** выбирается shift.
- При конфликтах **reduce/reduce** выбирается правило, по которому входной поток может быть сокращен раньше.

Использование действий внутри правил может привести к конфликтам в тех случаях, когда действие должно быть выполнено до того, как анализатор определит правило. В таких ситуациях использование приведенных

выше правил создает ошибочный анализатор. По этой причине программа **yacc** сообщает о числе конфликтов **shift/reduce** и **reduce/reduce**, разрешенных с использованием предыдущих правил.

Применение отладочного режима в анализаторе, созданном yacc

Вы можете получить доступ к отладочному коду, вызывав команду **yacc** с опцией **-t** или откомпилировав файл **y.tab.c** с ключом **-DYYDEBUG**.

В обычном режиме работы внешняя переменная *yydebug* имеет значение 0. Однако, если ей присвоено ненулевое значение, анализатор будет создавать описания входных лексем и действий, предпринимаемых при обработке входного потока.

Для установки переменной можно использовать несколько способов:

- Добавить следующую строку в раздел объявлений файла грамматики **yacc**:
  ```
  int yydebug = 1;
  ```
- Запускать анализатор с помощью программы **dbx** и изменять значение переменной с помощью команд **dbx**.

Примеры программ с использованием lex и yacc

В этом разделе приведены примеры программ для команд **lex** и **yacc**.

Эти программы описывают простой калькулятор, поддерживающий операции сложения, вычитания, умножения и деления. Калькулятор также позволяет присваивать значения переменным (обозначаемым одной строчной буквой) и затем использовать их в вычислениях. Примеры программ **lex** и **yacc** находятся в следующих файлах:

Файл	Содержание
calc.lex	Файл спецификаций **lex**, в котором определяются правила лексического анализа.
calc.yacc	Файл грамматики **yacc**, который содержит правила грамматического разбора. Для ввода информации применяется функция **yylex**, созданная командой **lex**.

Везде далее предполагается, что примеры программ **calc.lex** и **calc.yacc** расположены в текущем каталоге.

Компиляция примеров

Для создания программы, реализующей компилятор, выполните следующие действия:

1. Обработайте файл грамматики **yacc**, указав флаг **-d** (в результате **yacc** создаст не только программу на C, но и файл определений лексем):
   ```
   yacc -d calc.yacc
   ```
2. С помощью команды **ls** убедитесь, что были созданы следующие файлы:

   y.tab.c Исходный файл на языке C, созданный командой **yacc** для синтаксического анализатора

   y.tab.h Заголовочный файл, содержащий определения лексем, используемых анализатором
3. Обработайте файл спецификаций **lex**:
   ```
   lex calc.lex
   ```
4. С помощью команды **ls** убедитесь, что был создан следующий файл:

   lex.yy.c
   Исходный файл на языке C, созданный командой **lex** для лексического анализатора
5. Скомпилируйте и скомпонуйте два исходных файла на языке C:
   ```
   cc y.tab.c lex.yy.c
   ```
6. С помощью команды **ls** убедитесь, что были созданы следующие файлы:

   y.tab.o Объектный файл для исходного файла **y.tab.c**

lex.yy.o
Объектный файл для исходного файла **lex.yy.c**

a.out Исполняемая программа

Для того чтобы запустить программу **a.out**, введите:

```
$ a.out
```

ИЛИ

Для того чтобы переименовать файл и запустить программу, введите:

```
$ mv a.out calculate
$ calculate
```

В любом случае, после запуска программы курсор будет помещен на строку, расположенную ниже приглашения командной строки ($). После этого введите цифры и арифметические операторы, как и в любом другом калькуляторе. После нажатия клавиши Enter программа выведет результаты операции. Если вы присвоите значение переменной, как показано ниже, курсор переместится на следующую строку.

```
m=4 <enter>
_
```

После этого вы сможете использовать переменную в дальнейших вычислениях:

```
m+5 <enter>
9
_
```

Исходный текст синтаксического анализатора

Ниже приведен текст файла **calc.yacc**. В этом файле есть код из всех трех разделов файла грамматики **yacc**: объявления, правила и программы.

```
%{
#include<stdio.h>

int regs[26];
int base;

%}

%start list

%union { int a; }


%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*унарная операция вычитания */

%%                   /* начало раздела правил */

list:                        /*empty */
         |
        list stat '\n'
         |
        list error '\n'
         {
           yyerrok;
         }
```

```
        ;
stat:   expr
        {
          printf("%d\n",$1);
        }
        |
        LETTER '=' expr
        {
          regs[$1.a] = $3.a;
        }

        ;

expr:   '(' expr ')'
        {
          $$ = $2;
        }
        |
        expr '*' expr
        {

          $$.a = $1.a * $3.a;
        }
        |
        expr '/' expr
        {
          $$.a = $1.a / $3.a;
        }
        |
        expr '%' expr
        {
          $$.a = $1.a % $3.a;
        }
        |
        expr '+' expr
        {
          $$.a = $1.a + $3.a;
        }
        |
        expr '-' expr
        {
          $$.a = $1.a - $3.a;
        }
        |
        expr '&' expr
        {
          $$.a = $1.a & $3.a;
        }
        |
        expr '|' expr
        {
          $$.a = $1.a | $3.a;
        }
        |

       '-' expr %prec UMINUS
        {
          $$.a = -$2.a;
        }
        |
        LETTER
        {
          $$.a = regs[$1.a];
        }

        |
        number
```

```
        ;

number:  DIGIT
         {
           $$ = $1;
           base = ($1.a==0) ? 8 : 10;
         }       |
         number DIGIT
         {
           $$.a = base * $1.a + $2.a;
         }
         ;

%%
main()
{
 return(yyparse());
}

yyerror(s)
char *s;
{
  fprintf( stderr, "%s\n" ,s);
}

yywrap()
{
  return(1);
}
```

Файл содержит следующие разделы:

- **Раздел объявлений.** Записи этого раздела выполняют следующие функции:
  - Подключают заголовочный файл стандартных функций ввода-вывода
  - Определяют глобальные переменные
  - Определяют правило `list` как начальное правило
  - Определяют лексемы, используемые анализатором
  - Определяют операторы и их приоритет
- **Раздел правил.** Раздел правил содержит правила грамматического разбора входного потока.
  - ***%start*** - Указывает, что ввод должен совпадать с **stat**.
  - ***%union*** - По умолчанию действия и лексический анализатор возвращают целые числа. **yacc** может поддерживать значения других типов, включая структуры. Кроме того, **yacc** отслеживает типы и добавляет имена элементов объединения с учетом типов анализатора. Стек значений **yacc** объявляется в виде объединения с различными типами требуемых значений. Пользователь объявляет объединение и связывание имена элементов объединения с маркерами и нетерминальными символами со значениями. Если значение указано с помощью конструкции $$ или $n, то **yacc** автоматически добавляет имя соответствующего объединения во избежание нежелательных преобразований.
  - ***%type*** - Позволяет использовать элементы объявления **%union** и указывает отдельный тип для значений, связанных с каждым элементом грамматики.
  - ***%toksn*** - выдает список маркеров из инструмента lex со связанными типами.
- **Раздел программ.** Раздел программ содержит описанные ниже функции. В связи с тем, что все необходимые функции определены в самом файле, при его обработке не требуется подключать библиотеку **yacc**.

Функция	Описание
main	Функция, с которой начинается выполнение программы. Она вызывает функцию **yyparse** для запуска анализатора.
yyerror(s)	Эта функция обработки ошибок выводит сообщения об ошибках.
yywrap	Функция, которая возвращает 1 при обнаружении конца ввода.

Исходный текст лексического анализатора

В этом файле подключается библиотека стандартного ввода-вывода и файл **y.tab.h**. Если в команде **yacc** указан флаг **-d**, то программа **yacc** создает этот файл из информации, содержащейся в файле грамматики **yacc**. Файл **y.tab.h** содержит определения лексем, используемых синтаксическим анализатором. Файл **calc.lex** содержит правила формирования этих лексем из данных ввода. Ниже приведен текст файла **calc.lex**.

```
%{

#include<stdio.h>
#include "y.tab.h"
int c;
%}
%%
" "       ;
[a-z] {
            c = yytext[0];
            yylval.a = c - 'a';
            return(LETTER);
          }
[0-9] {
            c = yytext[0];
            yylval.a = c - '0';
            return(DIGIT);
          }
[^a-z0-9\b] {
                 c = yytext[0];
                 return(c);
               }
%%
```

Команда make

В этом разделе рассказано, как можно упростить процедуру повторной компиляции и компоновки с помощью команды **make**.

Связи между файлами достаточно определить только один раз. После этого команда **make** будет автоматически выполнять обновление.

В любом проекте программы обычно компонуются из объектных файлов и библиотек. Следовательно, после изменения исходного файла вам необходимо заново перекомпилировать часть исходных модулей и заново перекомпоновать программу. Команда **make** применяется для обслуживания набора программ, которые обычно относятся к одному проекту, путем добавления новых версий программ. Команда **make** особенно полезна при создании программ среднего размера. Данная программа не предназначена для решения задач, связанных с поддержкой нескольких версий исходного кода и с описанием больших программ (см. описание команды **sccs**).

С помощью команды **make** можно выполнить следующие задачи:

- Сохранить инструкции по созданию большой программы в одном файле.
- Задать макроопределения в файле описания команды **make**.
- Определить способ создания файла с помощью команд оболочки, либо создать набор базовых типов файлов с помощью команды **make**.
- Создать библиотеки.

Для работы команде **make** необходимы файл описания, список имен файлов, а также правила, по которым команда **make** должна создавать стандартные типы файлов, и временные метки всех системных файлов.

Понятия, связанные с данным:

“Инструменты и утилиты” на стр. 1
В этом разделе приведен обзор инструментов и утилит, предназначенных для разработки программ на языке C.

Создание файла описания

Для создания целевого файла, содержащего весь исполняемый код, команда **make** использует информацию из файла описания (*целевого* файла).

На основе этой информации программа **make** определяет способ компоновки целевого файла, список файлов для компоновки и их связь с другими файлами. Файл описания содержит следующую информацию:

- Имя целевого файла
- Имена родительских файлов, на основе которых создается целевой файл
- Команды создания целевого файла из родительских
- Макроопределения в файле описания
- Пользовательские правила компоновки целевых файлов

Команда **make** проверяет временные метки родительских файлов и на основе этой информации составляет список файлов, которые необходимо заново создать для получения обновленной копии целевого файла. Если какой-либо родительский файл был изменен уже после создания целевого файла, то команда **make** заново создаст все файлы, связанные с родительским, в том числе целевой файл.

Если файлу описания присвоено имя **makefile** или **Makefile**, и этот файл расположен в целевом каталоге, то для обновления первого целевого файла и его родительских файлов введите следующую команду:

```
make
```

Файлы будут обновлены независимо от того, сколько из них было изменено с момента последнего создания целевого файла командой **make**. В большинстве случаев файл описания имеет простую структуру, и его не требуется часто обновлять.

Если вы собираетесь хранить в одном каталоге несколько файлов описаний, присвойте им разные имена. Команда

```
make -f файл-описания
```

где *файл-описания* - это имя файла описания.

Формат записи файла описания

Общий формат записи выглядит следующим образом:

```
целевой-файл-1[целевой-файл-2..]:[:][родительский-файл-1..][;команда]...
[(символ-табуляции) команды]
```

Элементы в квадратных скобках - необязательные. Имена целевых и родительских файлов - это символьные строки, которые могут содержать буквы, цифры, точки и символы косой черты. Команда **make** позволяет указывать символы подстановки, такие как * (звездочка) и ? (знак вопроса). Строка файла описания, содержащая имя целевого файла, называется *строкой взаимосвязей*. Строки, содержащие команды, должны начинаться с символа табуляции.

> **Примечание:** в команде **make** символ $ (знак доллара) применяется для обозначения макроопределения. Не используйте этот символ в именах целевых или родительских файлов и в командах файла описания, если вы не применяете макроопределение **make**.

Комментарии в файле описания должны начинаться со знака фунта (#). Команда **make** игнорирует символ # и все следующие за ним символы. Кроме того, команда **make** игнорирует пустые строки.

Длина всех строк в файле описания, за исключением строк комментариев, может превышать ширину строки устройства ввода. Для переноса строки укажите в позиции переноса символ \ (обратная косая черта).

Применение команд в файле описания make

Команда - это любая строка символов, кроме строк, начинающихся с символа # (знака фунта) или с символа новой строки. Символ # можно использовать в команде только в кавычках.

Команды могут задаваться либо в строке взаимосвязей после двоеточия, либо в строках, начинающихся с символа табуляции, которые следуют непосредственно за строкой взаимосвязей.

В файле описания для целевого файла может быть задана одна последовательность команд, либо несколько последовательностей для различных наборов зависимостей. Оба способа задания команд одновременно использовать нельзя.

Если для создания целевого файла будет применяться одна последовательность команд, укажите одно двоеточие после имени целевого файла в строке взаимосвязей. Пример:

```
test:        список-взаимосвязей-1...
         список-команд...
            .
            .
            .
test:        список-взаимосвязей-2...
```

задается имя целевого файла (**test**), набор родительских файлов и набор команд для создания файла. Для целевого файла **test** может быть задан и другой список взаимосвязей. Однако для этого имени нельзя будет указать другой список команд. Если будет изменен один из файлов, от которых зависит файл **test**, команда **make** создаст целевой файл **test**, выполнив команды из заданного списка.

Для задания нескольких последовательностей команд для создания одного и того же файла определите несколько списков взаимосвязей. В каждой строке взаимосвязей должно быть задано имя целевого файла, после которого должны быть указаны два двоеточия (::), список взаимосвязей и список команд, которые нужно выполнить команде **make** при изменении каких-либо файлов в списке взаимосвязей. Пример:

```
test::      список-взаимосвязей-1...
         список-команд-1...
test::      список-взаимосвязей-2...
         список-команд-2...
```

Этот файл определяет два способа создания целевого файла с именем **test**. При изменении файлов из `списка-взаимосвязей-1` команда **make** выполняет команды из `списка-команд-1`. При изменении файлов из `списка-взаимосвязей-2` команда **make** выполняет команды из `списка-команд-2`. Во избежание конфликтов родительский файл не должен присутствовать ни в одном из списков взаимосвязей.

> **Примечание:** команда **make** передает команды из каждой командной строки в новую оболочку. Будьте внимательны при использовании команд, которые зависят от процесса оболочки, например, команды **cd** и команд оболочки. Команда **make** не сохранит результат выполнения этих команд до выполнения команды, указанной в следующей строке файла описания.

Для объединения двух строк файла в одну командную строку, укажите в конце первой строки файла символ \ (обратную косую черту). Команда **make** объединит две строки в одну и передаст их в одну и ту же оболочку.

Вызов команды make из файла описания

Для создания вложенного вызова команды **make** в файле описания укажите в командной строке этого файла макроопределение **$(MAKE)**.

Если указан флаг **-n** и было найдено макроопределение **$(MAKE)**, то новая копия команды **make** не будет выполнять никаких команд, за исключением другой команды макроопределения **$(MAKE)**. Для применения этой возможности для тестирования набора файлов описания программы, введите следующую команду:

```
make -n
```

Эта команда **make** не выполняет никаких операций с программой. Однако она должна записать в стандартный вывод сведения обо всех этапах компиляции программы, в том числе вывод низкоуровневых вызовов команды **make**.

Предотвращение остановки выполнения команды make из-за ошибки

Если какая-либо программа передаст ненулевой код возврата, то команда **make** обычно завершает работу. Некоторые программы возвращают код возврата, который не сигнализирует об ошибке.

Для того чтобы предотвратить завершение программы **make** при возникновении ошибки, выполните одно из следующих действий:

- Введите команду **make** с флагом **-i**.
- Задайте в строке взаимосвязей файла описания имя фиктивного целевого файла **.IGNORE**. Поскольку целевой файл **.IGNORE** реально не существует, он называется фиктивным. Если вместе с **.IGNORE** указаны предварительные условия, то связанные с ними ошибки команда **make** будет игнорировать.
- Укажите знак - (минус) в первой позиции всех строк файла описания, ошибки в которых должны игнорироваться.

Пример файла описания

Допустим, после компиляции и компоновки трех файлов на языке C (**x.c**, **y.c** и **z.c**) создается программа с именем **prog**. Общие объявления для файлов **x.c** и **y.c** заданы в файле с именем `defs`. В файле **z.c** эти объявления не используются. Ниже приведен пример файла описания, который создает программу **prog**:

```
# Создание программы prog из трех объектных файлов
prog:  x.o y.o z.o
# Создание prog с помощью cc
    cc  x.o y.o z.o -o prog
# Создать x.o из двух других файлов
x.o:   x.c defs
# Создание x.o с помощью cc
    cc -c x.c
# Создание y.o из двух других файлов
y.o: y.c defs
# Создание y.o с помощью cc
    cc  -c y.c
# Создание z.o из z.c
z.o:   z.c
# Создание z.o с помощью cc
    cc  -c z.c
```

Если этот файл называется **makefile**, введите команду **make** для обновления программы **prog** после внесения изменений в один из исходных файлов: **x.c**, **y.c**, **z.c** или **defs**.

Упрощение файла описания

Для упрощения файла описания можно воспользоваться внутренними правилами программы **make**.

В соответствии с соглашениями о присвоении имен в файловой системе, команда **make** распознает три файла **.c**, соответствующие нужным файлам **.o**. С помощью команды **cc -c** эта программа может создать объектный файл из исходного.

С учетом этих правил файл описания примет следующий вид:

```
# Создание программы prog из трех объектных файлов
prog:  x.o y.o z.o
# Создание prog с помощью cc
```

```
    cc  x.o y.o z.o -o prog
# Использование файлов defs и .c
# для создании x.o и y.o
x.o y.o:   defs
```

Внутренние правила программы make

Внутренние правила команды **make** хранятся в файле, аналогичном файлам описания.

Если указать флаг **-r**, команда **make** не будет использовать файл внутренних правил. В этом случае необходимо определить правила создания файлов в файле описания. Файл внутренних правил содержит список расширений имен файлов (таких как **.o** или **.a**), которые распознаются командой **make**, и правила, указывающие команде **make** способ создания файла с одним расширением из файла с другим расширением. По умолчанию команда **make** распознает следующие расширения имен:

Суффикс	Описание
.a	Архивная библиотека
.C	Исходный файл на языке C++
.C\~	Файл системы управления исходным кодом (SCCS), содержащий исходный файл на языке C++
.c	Исходный файл на языке C
.c~	Файл SCCS, содержащий исходный файл на языке C
.f	Исходный файл на языке FORTRAN
.f~	Файл SCCS, содержащий исходный файл на языке FORTRAN
.h	Заголовочный файл на языке C
.h~	Файл SCCS, содержащий заголовочный файл на языке C
.l	Исходная грамматика **lex**
.l~	Файл SCCS, содержащий исходную грамматику **lex**
.o	Объектный файл
.s	Исходный файл на ассемблере
.s~	Файл SCCS, содержащий исходный файл на ассемблере
.sh	Исходный файл команд оболочки
.sh~	Файл SCCS, содержащий исходный файл команд оболочки
.y	Исходная грамматика **yacc-c**
.y~	Файл SCCS, содержащий исходную грамматику **yacc-c**

Список расширений имен файлов аналогичен списку взаимосвязей в файле описания и следует за фиктивным именем целевого файла **.SUFFIXES**. Команда **make** просматривает список расширений имен файлов слева направо, поэтому последовательность записей существенна.

Команда **make** будет использовать первую же запись из списка, которая удовлетворяет следующим требованиям:

- Запись совпадает с требованиями к расширению имени входного или выходного файла для текущих целевого файла и файла зависимостей.
- С данной записью связано правило.

Из двух расширений имен файлов, определяемых правилом, команда **make** создает имя правила. Например, имя правила для преобразования файла **.c** в файл **.o** - **.c.o**.

Для того чтобы добавить в список другие расширения имен файлов, добавьте в файл описания запись для фиктивного целевого имени **.SUFFIXES**. Если задать в файле описания строку **.SUFFIXES** без расширений имен файлов, команда **make** сотрет текущий список. Для изменения порядка имен в списке сотрите текущий список, а затем укажите в строке **.SUFFIXES** новый набор значений.

Пример файла правил по умолчанию

Ниже приведен фрагмент файла правил, используемых по умолчанию:

```
# Определение расширений имен файлов, известных команде make.
.SUFFIXES:  .o .C .C\~ .c .c~ .f .f~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .a
 # Начало списка внутренних
```

```
# макроопределений
YACC=yacc
YFLAGS=
ASFLAGS=
LEX=lex
LFLAGS=
CC=cc
CCC=xlC
AS=as
CFLAGS=
CCFLAGS=
# Конец списка внутренних
# макроопределений
# Создание файла .o из файла .c
# с помощью программы cc.
c.o:
        $(CC) $(CFLAGS) -c $<
# Создание файла .o
# из файла .s с помощью ассемблера.
s.o:
         $(AS)$(ASFLAGS) -o $@ $<
.y.o:
# Создание промежуточного файла с помощью yacc
         $(YACC) $(YFLAGS) $<
# Вызов компилятора cc
         $(CC) $(CFLAGS) -c y.tab.c
# Удаление промежуточного файла
         rm y.tab.c
# Копирование в целевой файл
         mv y.tab.o $@.
.y.c:
# Создание промежуточного файла с помощью yacc
         $(YACC) $(YFLAGS) $<
# Копирование в целевой файл
         mv y.tab.c $@
```

Правила с одним суффиксом

В команде **make** существует набор правил с одним расширением, позволяющий создавать из исходного файла целевой файл без расширения (например, командный файл).

Существуют также правила, в соответствии с которыми команда **make** создает объектные файлы с именами без расширений из следующих исходных файлов:

Команда	Описание
.C:	Из исходного файла на языке C++
.C\~:	Из исходного файла SCCS на языке C++
.c:	Из исходного файла на языке C
.c~:	Из исходного файла SCCS на языке C
.sh:	Из файла команд оболочки
.sh~:	Из файла SCCS, содержащего команды оболочки

Например, если все необходимые исходные файлы расположены в одном каталоге, то для создания программы `cat` введите следующую команду:

```
make cat
```

Создание библиотек с помощью команды make

В команде **make** существует набор правил с одним расширением, позволяющий создавать из исходного файла целевой файл без расширения (например, командный файл).

Существуют следующие внутренние правила преобразования исходных файлов в библиотечные файлы:

Правило	Описание
.C.a	Исходный файл C++ в архивный
.C\~.a	Исходный файл C++ SCCS в архивный
.c.a	Исходный файл C в архивный
.c~.a	Исходный файл C SCCS в архивный
.s~.a	Исходный ассемблерный файл SCCS в архивный
.f.a	Исходный файл Fortran в архивный
.f~.a	Исходный файл Fortran SCCS в архивный

Определение команд по умолчанию в файле описания

Если при создании целевого файла команда **make** не может найти в файле описания необходимые команды или внутренние правила, она применяет указанные в файле команды по умолчанию.

Команды, которые команда **make** будет выполнять в этом случае, перечисляются после целевого имени **.DEFAULT** следующим образом:

```
.DEFAULT:
                команда
                команда
                   .
                   .
                   .
```

Поскольку целевой файл **.DEFAULT** реально не существует, он называется *фиктивным*. Фиктивный целевой файл **.DEFAULT** применяется для задания функции исправления ошибок или общей процедуры, которая применяется для создания всех файлов, не определенных во внутренних правилах команды **make**.

Добавление других файлов в файл описания

С помощью директивы **include** в файл описания можно включать другие файлы. Эту директиву можно указывать в любой строке файла.

После слова include указывается пробел или символ табуляции, за которым должно следовать имя файла, который будет включен командой **make**.

Примечание: В директиве **include** можно задать только одно имя файла.

Пример:

```
include /home/tom/temp
include /home/tom/sample
```

указывают команде **make**, что для создания целевого файла необходимо использовать файлы **temp**, **sample** и файл описания.

Файлы описания не следует использовать с уровнем вложенности больше 16.

Определение и использование макроопределений в файле описания

Макроопределение задает имя или метку, которая будет использоваться вместо нескольких других имен. Макроопределения позволяют кратко записывать длинные выражения.

Для создания макроопределения:

1. В начале новой строки введите имя макроопределения.
2. После имени введите знак равенства (=).
3. Справа от знака равенства (=) введите строку символов, которую должно заменять указанное имя.

До и после знака равенства (=) в макроопределении могут стоять пробелы; они ни на что не влияют. Перед знаком равенства не должно стоять двоеточие (:) или символ табуляции.

Ниже приведены примеры макроопределений:

```
# С константой "2" связывается значение "xyz"
2 = xyz

# С константой "abc" связывается значение "-ll -ly"
abc = -ll -ly

# С константой "LIBES" связывается пустое значение
LIBES =
```

Если задано имя макроопределения, но не указано его значение, то оно считается пустым.

Использование макроопределений в файле описания

После того как вы зададите макроопределение в файле описания, имя этого макроопределения можно использовать в командах файла описания, указывая перед ним символ $ (знак доллара).

Если длина имени больше одного символа, его необходимо заключить в круглые () или фигурные {} скобки. Ниже приведены примеры использования макроопределений:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

Последние два примера в этом списке эквивалентны.

Ниже приведен пример файла описания, в котором задаются и используются макроопределения:

```
# OBJECTS - это три файла x.o, y.o и
# z.o (откомпилированные ранее)
OBJECTS = x.o y.o z.o
# LIBES - это библиотека стандартных функций
LIBES = -lc
# prog зависит от x.o y.o и z.o
prog: $(OBJECTS)
# Компоновка и загрузка 3 файлов и библиотеки стандартных
# функций для создания файла prog
      cc $(OBJECTS) $(LIBES) -o prog
```

С помощью этого файла описания команда **make** компонует и загружает три объектных файла (**x.o**, **y.o** и **z.o**) и библиотеку **libc.a**.

Макроопределение, которое вводится в командной строке, переопределяет макроопределение с тем же именем, заданное в файле описания. Таким образом, следующая команда загружает файлы и библиотеку **lex**(`-ll`):

```
make "LIBES= -ll"
```

> **Примечание:** Если внутри макроопределения, указанного в командной строке, есть пробелы, его необходимо заключить в двойные кавычки (" "). В противном случае оболочка будет воспринимать эти пробелы не как часть макроопределения, а как разделители параметров.

Команда **make** поддерживает до 10 уровней вложенности макроопределений. В соответствии с определениями из следующего примера выражение `$($(macro2))` будет равно `value1`:

```
macro1=value1

macro2=macro1
```

Значение в макроопределении вычисляется в момент его обработки. Оно не вычисляется в момент определения. Если макроопределение было задано, но ни разу не использовалось, то связанное с ним значение не будет ни разу вычислено. Это особенно важно в тех случаях, когда макроопределение создается для значений, интерпретируемых оболочкой, так как они могут изменяться. Результат подстановки макроопределения

```
OBJS = 'ls *.o'
```

может быть разным, если оно вызывается в разные моменты времени в процессе создания или удаления объектных файлов. Результат обработки такого макроопределения не совпадает с результатом вызова команды **ls** во время создания макроопределения **OBJS**.

Стандартные макроопределения

В команде **make** предусмотрены стандартные макроопределения, которые можно использовать в файле описания.

В частности, они позволяют задавать переменные. Команда **make** выполняет следующие макроподстановки:

Макрокоманда	Значение
$@	Имя текущего целевого файла
$$@	Имя метки в строке взаимосвязей
$?	Имена устаревших файлов (т.е. тех, которые были изменены уже после создания целевого файла)
$<	Имя измененного родительского файла, из-за которого требуется повторно создать целевой файл
$*	Имя текущего родительского файла без расширения
$%	Имя компонента архивной библиотеки

Имя целевого файла

Если в последовательности команд файла описания задано имя **$@**, то перед тем, как передать очередную команду оболочке, программа **make** заменит это имя на имя текущего целевого файла. Команда **make** выполняет такую подстановку только в том случае, если при создании целевого файла она выполняет команды из файла описания.

Имя метки

Если в строке взаимосвязей файла описания указано имя **$$@**, то программа **make** подставляет вместо него имя метки, которое стоит в этой строке слева от двоеточия. Например, строку взаимосвязей

```
cat:     $$@.c
```

команда **make** преобразует в

```
cat:     cat.c
```

после обработки. Это макроопределение можно использовать для создания группы файлов, каждому из которых соответствует только один исходный файл. Например, для обновления каталога системных команд можно создать такой файл описания:

```
# Макроопределение CMDS
# для замены имен команд
CMDS = cat dd echo date cc cmp comm ar ld chown
# Каждая команда зависит от файла .c
$(CMDS):      $$@.c
# Создание нового набора команд путем компиляции устаревших
# файлов ($?) и создания целевого файла с именем ($@)
         $(CC) -O $? -o $@
```

При выполнении команды **make** вместо **$$(@F)** будет подставлено **$@**. Это макроопределение можно использовать для обновления каталога **usr/include**, когда файл описания находится в другом каталоге. Ниже приведен пример такого файла описания:

```
# Макроопределение для имени каталога INCDIR
INCDIR = /usr/include
# Замена группы файлов каталога
# именем INCLUDES
INCLUDES = \
         $(INCDIR)/stdio.h \
         $(INCDIR)/pwd.h \
         $(INCDIR)/dir.h \
         $(INCDIR)/a.out.h \
# Все файлы в списке зависят от файла
# с тем же именем из текущего каталога
$(INCLUDES):       $$(@F)
# Копирует более свежие файлы из текущего
# каталога в /usr/include
         cp $? $@
# Устанавливает для целевых файлов права только на чтение
         chmod 0444 $@
```

Предыдущий файл описания создает файл в каталоге **/usr/include**, если был изменен соответствующий файл в текущем каталоге.

Измененные файлы

Если в последовательности команд файла описания есть макроопределение **$?**, то команда **make** подставляет вместо него список родительских файлов, которые были изменены уже после создания целевого файла. Команда **make** выполняет такую подстановку только в том случае, если при создании целевого файла она выполняет команды из файла описания.

Первый устаревший файл

Если в последовательности команд файла описания задано макроопределение **$<**, то команда **make** подставляет вместо него имя файла, с которого начинается создание целевого файла. Это имя родительского файла, который был изменен после создания целевого файла, что и послужило причиной вызова команды **make**.

Если указать символ **<** (знак "меньше") с буквой **D** или **F**, то вместо этой конструкции будет подставлено имя каталога или имя первого устаревшего файла, соответственно. Например, если имя первого устаревшего файла

```
/home/linda/sample.c
```

то в результате подстановки, выполняемой командой **make**, он будет заменен на

```
$(<D)   =   /home/linda
$(<F)   =   sample.c
$<      =   /home/linda/sample.c
```

Команда **make** выполняет эту подстановку только в том случае, если она выполняет команды из своих внутренних правил или из списка **.DEFAULT**.

Расширение текущего имени файла

Если в последовательности команд файла описания задано макроопределение **$***, то команда **make** подставляет вместо него имя текущего родительского файла (без расширения), который применяется для создания целевого файла. Например, если команда **make** использует файл

```
test.c
```

то вместо **$*** будет подставлено имя `test`.

Если после звездочки (*) указать букву **D** или **F**, то будет подставляться имя каталога или имя текущего файла, соответственно.

Допустим, что для создания целевого файла программа **make** использует несколько файлов (заданных в файле описания или во внутренних правилах). В каждый момент времени применяется только один из этих файлов (он называется текущим). Если этот текущий файл

```
/home/tom/sample.c
```

то результат подстановки, выполняемой командой **make**, будет выглядеть так:

```
$(*D)  =  /home/tom
$(*F)  =  sample
$*     =  /home/tom/sample
```

Команда **make** выполняет эту подстановку только в том случае, если она выполняет команды из своих внутренних правил или из списка **.DEFAULT**, а не из файла описания.

Компонент архивной библиотеки

Если в последовательности команд файла описания задано макроопределение **$%**, а целевой файл представляет собой компонент архивной библиотеки, то программа **make** подставляет имя библиотечного компонента. Например, если целевой файл - это

```
lib(file.o)
```

то команда **make** подставляет вместо **$%** имя компонента, то есть `file.o`.

Изменение макроопределений в команде

Если в командах оболочки файла описания применяются макроопределения, то вы можете изменить текст, который будет подставлен вместо них командой **make**.

Для изменения текста замещения после имени макроопределения нужно поставить двоеточие (:), а затем указать новый текст. Ниже приведен общий вид этой команды:

```
$(имя:строка1=строка2)
```

где `строка1` - заменяемый суффикс или слово, а `string2` - заменяющий суффикс или слово.

Когда команда **make** обнаруживает имя макроопределения, вместо `строки1` она подставляет `строку2`. Например, если файл описания содержит макроопределение

```
FILES=test.o sample.o form.o defs
```

то с помощью макроопределения в команде файла описания можно заменить файл `form.o` на `input.o`:

```
cc -o $(FILES:form.o=input.o)
```

В макроопределении можно заменить все суффиксы .o на .c:

```
cc -c $(FILES:.o=.c)
```

Значения в макроопределении можно также изменить по шаблону следующим образом:

```
$(macro: op%os= np%ns)
```

Здесь `op` - старый префикс, `os` - старый суффикс, `np` - новый префикс, а `ns` - новый суффикс.

Значения `op`, `os`, `np` и `ns` должны быть строками произвольной длины (включая нулевую). Шаблон, соответствующий знаку процентов (%) слева от знака равенства (строка из 0 или более символов), применяется с `np` и `ns` для замены значения макроопределения. Оператор % может быть указан произвольное число раз справа от знака равенства (=).

Например:

```
FOO=abc def
BAR=$(FOO:%=dir1/%.o dir1/%_cltn.o)
```

Присваивает BAR значения `dir1/abc.o dir1/abc_cltn.o dir1/def.o dir1/def_cltn.o`

Такое изменение макроопределения можно использовать при обновлении архивных библиотек. Дополнительная информация приведена в описании команды **ar**.

Создание целевых файлов с помощью команды make

Программа **make** создает целевой файл следующим образом:

1. Находит имя целевого файла в файле описания или в команде вызова программы **make**
2. Проверяет, существуют ли файлы, от которых зависит целевой файл, и не устарели ли они
3. Проверяет, совпадает ли версия целевого файла с версией файлов, от которых он зависит.

Если целевой файл или один из родительских файлов устарел, команда **make** создает целевой файл с помощью одного из следующих наборов команд:

- Команды из файла описания
- Внутренние правила создания файла (если они применимы)
- Команды по умолчанию из файла описания

Если все файлы уже были обработаны программой **make**, то она выдаст сообщение о том, что ни один файл не изменен, и завершит свою работу. Если после последнего запуска команды **make** некоторые файлы были изменены, то команда make обновит только устаревшие файлы. Файлы, которые не были изменены, заново не создаются.

При выполнении команд, предназначенных для создания целевого файла, команда **make** обрабатывает макроопределения, записывает каждую командную строку и передает команду в новую копию оболочки.

Применение команды make к файлам системы контроля исходного кода

Команды и файлы SCCS применяются для управления доступом к файлу и отслеживания изменений, внесенных в файл.

Файл SCCS - это любой текстовый файл, который применяется командами SCCS. При обработке файлов SCCS командами, отличными от SCCS, эти файлы могут быть повреждены.

У всех файлов SCCS есть префикс **s.**, отличающий их от обычных текстовых файлов. Команда **make** не распознает ссылки на префиксы имен файлов. Из-за этого в файле описания команды **make** нельзя ссылаться на файлы SCCS напрямую. Для представления файлов SCCS команда **make** использует расширение ~ (тильду). Следовательно, **.c~.o** - это имя правила, которое преобразует файл SCCS, содержащий исходный код на языке C, в объектный файл. Ниже приведено внутреннее представление этого правила:

```
.c~.o:
        $(GET) $(GFLAGS) -p $< >$*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Добавление символа ~ (тильды) к расширению имени файла приводит к тому, что вместо обычного файла будет найден файл SCCS с фактическим расширением (включающим все символы, стоящие между точкой (.) и тильдой). С помощью макроопределения **GFLAGS** в SCCS передаются флаги, задающие версию файла SCCS.

Команда **make** распознает следующие расширения имен SCCS:

Суффикс	Описание
.C\~	Исходный файл C++
.c~	Исходный файл **c**
.y~	Исходная грамматика **yacc**
.s~	Исходный файл ассемблера
.sh~	Файл команд оболочки
.h~	Заголовочный файл
.f~	Файл кода на языке Fortran
.l~	Исходная грамматика **lex**

В команде **make** применяются следующие правила преобразования файлов SCCS:

.C\~.a:

.C\~.c:

.C\~.o:

.c~:

.c~.a:

.c~.c:

.c~.o:

.f~:

.f~.a:

.f~.o:

.f~.f:

.h~.h:

.l~.o:

.s~.a:

.sh~:

.s~.o:

.y~.c:

.y~.o:

Применение makefile в системе управления исходным кодом (SCCS) - Основные сведения

Если в текущем каталоге хранится файл описания (файл с именем **makefile** или **Makefile**), то команда **make** не будет искать файл описания в SCCS.

Если в текущем каталоге нет файла описания, то команда **make** попытается найти в SCCS файл с именем **s.makefile** или **s.Makefile**. Если команда **make** найдет один из этих файлов, она вызовет команду **get** для создания файла описания на базе найденного файла с помощью SCCS. После того как SCCS создаст файл описания, он будет обработан командой **make** как обычный файл описания. Когда команда **make** завершит свою работу, она удалит из текущего каталога созданный файл описания.

Отключение стандартного режима получения файлов из SCCS

Режим получения файлов из SCCS можно отключить. Команды получения исходных файлов из SCCS можно указать в правилах особого раздела *SCCS_GET* в файле описания.

Это позволяет переопределить стандартные правила извлечения исходных файлов из SCCS.

Например:

```
SCCS_GET:
        get -p $< > $*.c
```

Применение команды make другими файлами

Запустите команду **make** из каталога, в котором находится файл описания для создаваемого файла.

Имя этого файла описания указывается в параметре *файл-описания*. Введите команду

```
make -f файл-описания
```

Если файл описания называется **makefile** или **Makefile**, флаг **-f** указывать не нужно. Введите в командной строке вместе с командой **make** макроопределения, флаги, имена файлов описания и имена целевых файлов:

```
make [флаги]
[макроопределения] [целевые файлы]
```

С помощью параметров, заданных в командной строке, команда **make** определяет, что ей нужно сделать. Во-первых, она просматривает все макроопределения, указанные в командной строке (т.е. записи, которые заключены в кавычки и содержат знак равенства), и выполняет макроподстановку. Если команда **make** обнаружит одно и то же макроопределение в командной строке и в файле описания, то она будет применять макроопределение, заданное в командной строке.

После этого команда **make** просматривает флаги.

Все остальные записи в командной строке рассматриваются командой **make** как имена целевых файлов. Команда **make** выполняет все команды оболочки, заключенные в обратные кавычки, которые генерируют имена целевых файлов. После этого команда **make** создает целевые файлы (слева направо). Если имя целевого файла в командной строке не указано, то команда **make** создает целевой файл с первым же обнаруженным в файле описания именем, которое не начинается с точки. Если задано несколько файлов описания, команда **make** определяет имя целевого файла из первого файла описания.

Применение переменных среды командой make - основные сведения

При запуске программа **make** считывает текущие значения переменных среды и добавляет их в свои макроопределения.

С помощью макроопределений **MAKEFLAGS** и **MFLAGS** пользователь может задать флаги, которые должны быть переданы команде **make**. Если заданы оба макроопределения, то **MAKEFLAGS** переопределяет значения **MFLAGS**. С флагами, заданными с помощью этих переменных, в команду **make** передаются все опции из командной строки. Если команда **make** вызывается рекурсивно с помощью макроопределения **$(MAKE)** в файле описания, то при каждом вызове **make** передает все флаги.

Программа **make** выполняет макроподстановку в следующем порядке:

1. Считывает переменную среды **MAKEFLAGS**.

   Если переменная среды **MAKEFLAGS** не задана, команда **make** проверяет значение переменной среды **MFLAGS**. Если для одной из этих переменных задано непустое значение, команда **make** рассматривает каждый символ этого значения как входной флаг. Эти флаги (кроме флагов **-f**, **-p** и **-d**, которые нельзя задать с помощью **MAKEFLAGS** и **MFLAGS**), задают среду выполнения команды **make**.
2. Считывает и устанавливает флаги, заданные в командной строке. Эти флаги добавляются к флагам, установленным ранее с помощью переменной среды **MAKEFLAGS** или **MFLAGS**.
3. Считывает макроопределения, заданные в командной строке. Все другие макроопределения с таким же именем команда **make** будет игнорировать.
4. Считывает внутренние макроопределения.

5. Считывает значения переменных среды. Команда **make** рассматривает переменные среды как макроопределения и передает их в другие программы оболочки.

Применение команды make в режиме параллельного выполнения

Как правило, команда **make** одновременно обрабатывает только один целевой файл, последовательно выполняя отдельные команды и ожидая завершения текущей команды перед запуском следующей.

Однако команда **make** также поддерживает режим параллельного выполнения, позволяющий одновременно компоновать несколько независимых целевых файлов.

Флаг **-j** разрешает команде **make** одновременно обрабатывать несколько целевых файлов.

Если вместе с опцией **-j** указано целое число, оно задает максимальное число параллельных заданий, применяемых для компоновки целевых файлов.

Если флаг **-j** указан без целого числа, то число заданий не ограничено.

Если команда **make**, запущенная в режиме параллельного выполнения, в процессе компоновки одного из целевых файлов обнаружит ошибку и методы игнорирования ошибок не применяются, то дальнейшая обработка этого целевого файла будет приостановлена. Перед завершением команда make дождется выполнения всех запущенных дочерних заданий.

В режиме параллельного выполнения вывод нескольких команд выдается на экран по мере его поступления. Это может привести к путанице сообщений различных заданий. При необходимости вы можете подавить сообщения путем перенаправления или запустить команду **make** в режиме без вывода.

Макропроцессор m4 - обзор

В этом разделе описывается макропроцессор **m4**, который в операционной системе играет роль препроцессора для всех языков программирования.

В начале программы вы можете определить символьное имя или символьную константу как строку символов. Затем с помощью макропроцессора **m4** можно заменить все вхождения символьного имени (без кавычек) соответствующей строкой. Помимо замены одной строки текста на другую макропроцессор **m4** может выполнять следующие операции:

- Арифметические операции
- Операции с файлами
- Обработка условных макрокоманд
- Обработка строк и подстрок

Макропроцессор **m4** обрабатывает алфавитно-цифровые строки, которые называются *лексемами*. Макропроцессор **m4** считывает алфавитно-цифровую лексему и проверяет, не совпадает ли она с именем макроопределения. Если да, то программа подставляет вместо имени макрокоманды тело макроопределения и помещает полученную строку обратно во входной поток для повторного просмотра. При вызове макрокоманды можно указывать аргументы; в этом случае значения аргументов подставляются в тело макроопределения перед его повторным просмотром.

Макропроцессор **m4** обрабатывает стандартные макрокоманды, например, **define**. Кроме того, вы можете создать собственные макроопределения. Стандартные и пользовательские макрокоманды обрабатываются одинаково.

Работа с макропроцессором m4

Для вызова макропроцессора **m4** введите следующую команду:

```
m4 [файл]
```

Макропроцессор **m4** обрабатывает все аргументы по порядку. Если аргументы отсутствуют, либо указан аргумент **-** (дефис), **m4** считывает данные из стандартного ввода. Результаты работы макропроцессора **m4** отправляются в стандартный вывод. Если вы хотите сохранить их для дальнейшего использования, перенаправьте вывод в файл:

```
m4 [file] >файл-вывода
```

Создание пользовательских макроопределений

Макрокоманда	**Описание**
define (*имя*,*текст***)**	Создает новое макроопределение с заданным *именем* и со значением *текст-замещения*.

Например, если в программе задан оператор

```
define(name, stuff)
```

то макропроцессор **m4** присвоит строке `name` значение `stuff.` Если в файле программы встретится строка `name`, то макропроцессор **m4** подставит вместо нее строку `stuff.` Строка `name` должна быть алфавитно-цифровой строкой ASCII, начинающейся с буквы или символа подчеркивания. Вместо `stuff` можно задать любой текст; однако если этот текст содержит скобки, то число открывающих скобок, должно быть равно числу закрывающих скобок. Для переноса текста **stuff** на другую строку укажите символ / (косая черта).

Сразу за словом **define** должна стоять открывающая скобка. Пример:

```
define(N, 100)
 . . .
if (i > N)
```

определяет, что `N` равно `100`, после чего символьная константа N используется в операторе **if**.

Вызов макрокоманды в программе должен быть задан в следующем формате:

```
name(arg1,arg2, . . . argn)
```

Имя макроопределения считается таковым только в том случае, если оно не окружено алфавитно-цифровыми символами. В следующем примере переменная `NNN` никак не связана с макроопределением `N`.

```
define(N, 100)
 . . .
if (NNN > 100)
```

В макроопределениях можно использовать имена других макроопределений. Пример:

```
define(N, 100)
define(M, N)
```

определяет, что и `M`, и `N` равны `100`. Если затем изменить определение `N` и присвоить ей другое значение, то значение `M` по-прежнему останется равным `100`, а не `N`.

Как только макропроцессор **m4** встречает макрокоманду, он подставляет вместо нее текст замещения, указанный в макроопределении. Строка `N` заменяется на `100`. Затем строка `M` также заменяется на `100`. Результат не изменится, если сначала задать

```
define(M, 100)
```

а затем указать

```
define(M, N)
define(N, 100)
```

Теперь вместо M будет подставляться N, поэтому значение M всегда будет равно текущему значению N (в приведенном примере N равно 100).

Использование кавычек

Для того чтобы запретить замену аргументов **define**, их следует заключить в кавычки. По умолчанию для этого используются одинарные кавычки ` и ' (левая и правая кавычка). Для имени, указанного в кавычках, макроподстановка не выполняется. При подстановке самого имени кавычки удаляются. Значение строки в кавычках - это строка без кавычек. Например, если ввести

```
define(N, 100)
define(M, `N')
```

то при обработке аргумента удаляются кавычки вокруг N. За счет кавычек M определяется как строка N, а не как строка 100. В общем случае макропроцессор **m4** всегда удаляет один уровень кавычек при обработке лексемы. Это справедливо даже вне макроопределения. Для того чтобы в выводе команды сохранилось слово define, заключите его в кавычки:

```
`define' = 1;
```

Другой пример использования кавычек задает переопределение N (N указано в кавычках). Пример:

```
define(N, 100)
. . .
define(`N', 200)
```

Для того чтобы избежать ошибок, заключайте в кавычки первый аргумент макроопределения. Например, в следующем фрагменте программы аргумент N не будет переопределен:

```
define(N, 100)
. . .
define(N, 200)
```

N во втором определении заменяется на 100. Результат выполнения этих двух команд будет эквивалентен следующему оператору:

```
define(100, 200)
```

Макропроцессор **m4** игнорирует его, поскольку в качестве аргумента может быть задано только имя, но не число.

Переопределение кавычек

Обычно в качестве служебных символов применяются одинарные кавычки ` и ' (левая и правая). Если по каким-то причинам их использовать нельзя, замените их на другие символы с помощью стандартной макрокоманды

Макрокоманда	Описание
changequote (*l*, *r* **)**	Эта команда позволяет заменить левую и правую кавычки символами, указанными вместо переменных *l* и *r*.

Для того чтобы восстановить исходные функции кавычек, введите команду **changequote** без аргументов:

```
changequote
```

Аргументы

Простейший способ обработки макрокоманд - замена одной строки другой (фиксированной) строкой. Однако макрокоманда может содержать аргументы, позволяющие влиять на результат макрорасширения. Аргументы указываются в тексте замещения макроопределения (во втором аргументе) в формате **$***n* (*n* - номер аргумента). При обработке макрокоманды процессор **m4** подставляет вместо символа значение указанного аргумента. Например, символ

```
$2
```

обозначает второй аргумент макрокоманды. Следовательно, если задать следующее макроопределение с именем `bump`:

```
define(bump, $1 = $1 + 1)
```

то макропроцессор **m4** генерирует код для увеличения первого аргумента на 1. Выражение `bump(x)` равносильно выражению x = x + 1.

В макрокоманде можно указывать любое число аргументов. Однако с помощью конструкции **$***n* можно задать только 9 аргументов (с `$1` по `$9`). Если вам требуется большее число аргументов, воспользуйтесь макрокомандой **shift**.

Макрокоманда	**Описание**
shift (*список-параметров***)**	Возвращает все, за исключением первого, элементы списка *параметры*.

Эта макрокоманда удаляет первый аргумент и переприсваивает оставшиеся аргументы параметрам **$***n* (второй аргумент - параметру `$1`, третий - `$2`. . . десятый - `$9`). Выполнив несколько макрокоманд **shift**, можно перебрать все аргументы макроопределения.

Аргумент **$0** возвращает имя макроопределения. Если аргумент не указан в макрокоманде, он заменяется на пустую строку. Например, вы можете задать макроопределение, объединяющее аргументы следующим образом:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Так,

```
cat(x, y, z)
```

- то же самое, что и

```
xyz
```

Аргументы с `$4` по `$9` в этом примере представляют собой пустые строки, поскольку их значения не указаны.

Макропроцессор **m4** отбрасывает указанные без кавычек начальные пробелы, символы табуляции и символы новой строки, но сохраняет все остальные непечатаемые символы. Так,

```
define(a, b c)
```

определяет `a` как строку `b c`.

Аргументы разделяются запятыми. Если аргумент содержит запятые, его необходимо заключить в скобки, чтобы запятая не рассматривалась как разделитель. Например:

```
define(a, (b,c))
```

указано только два аргумента. Первый аргумент - `a`, второй - `(b,c)`. Для того чтобы в качестве значения задать запятую или круглую скобку, заключите ее в кавычки.

Стандартные макрокоманды m4

Макропроцессор **m4** предоставляет набор стандартных предопределенных макрокоманд. В этом разделе приведено описание этих макрокоманд.

Удаление макроопределений

Макрокоманда	Описание
undefine (`*имя*'**)**	Удаляет определение пользовательской или стандартной макрокоманды с указанным (`*именем*')

Пример:

```
undefine(`N')
```

удаляет определение N. Если с помощью **undefine** удалить стандартную макрокоманду, например:

```
undefine(`define')
```

то вы уже не сможете использовать эту макрокоманду.

Для того чтобы избежать подстановки, необходимо указывать имя в кавычках.

Проверка наличия макроопределения

Макрокоманда	Описание
ifdef (`*имя*'**,***аргумент-1***,***аргумент-2***)**	Если макроопределение с указанным *именем* существует, и его значение не равно нулю, то возвращает значение *аргумента1*. В противном случае возвращает *аргумент2*.

В макрокоманде **ifdef** предусмотрено три аргумента. Если первый аргумент определен, то значение **ifdef** равно второму аргументу. Если первый аргумент не определен, то значение **ifdef** равно третьему аргументу. Если третий аргумент не указан, **ifdef** возвращает пустое значение.

Арифметические операции над целыми числами

Макропроцессор **m4** предоставляет набор встроенных функций для выполнения арифметических действий над целыми числами:

Макрокоманда	Описание
incr (*число***)**	Увеличивает *число* на 1.
decr (*число***)**	Уменьшает *число* на 1.
eval	Вычисляет арифметическое выражение.

Следовательно, для того чтобы определить переменную, значение которой на единицу больше, чем заданное *Число*, нужно ввести:

```
define(Number, 100)
define(Number1, `incr(Number)')
```

В этом примере определяется число `Number1`, значение которого на единицу больше, чем значение `Number`.

Функция **eval** может вычислять значения выражений, содержащих следующие операторы (перечислены в порядке убывания приоритета):

- унарные **+** и **-**
- ****** и **^** (возведение в степень)
- *** / %** (модуль)
- **+ -**
- **== != < <= > >=**
- **!**(не)
- **&** и **&&** (логическое И)
- **|** и **||** (логическое ИЛИ)

Для выделения группы операций заключите их в скобки. В качестве операндов должны выступать числа. Истина (например, 1 > 0) имеет значение 1, ложь - 0. Точность функции **eval** составляет 32 разряда.

Например, с помощью функции **eval** можно создать макроопределение для M со значением 2==N+1:

```
define(N, 3)
define(M, `eval(2==N+1)')
```

Любой достаточно сложный текст замещения в макроопределении рекомендуется заключать в кавычки.

Операции с файлами

Для включения нового файла в программу предназначена встроенная функция **include**.

Макрокоманда	Описание
include (*файл***)**	Возвращает содержимое заданного *файла*.

Пример:

```
include(имя-файла)
```

подставляет содержимое файла имя-файла вместо команды **include**.

Если файл, имя которого указано в макрокоманде **include**, недоступен, возникает неустранимая ошибка. Для того чтобы ее избежать, используйте альтернативную форму **sinclude**.

Макрокоманда	Описание
sinclude (*файл* **)**	Возвращает содержимое указанного *файла*, но не выдает сообщения об ошибке, если *файл* недоступен.

Если файл недоступен, макрокоманда **sinclude** не выдает сообщение об ошибке, что позволяет программе продолжить свою работу.

Перенаправление вывода

Вывод макропроцессора **m4** можно перенаправить во временные файлы. После этого содержимое этих файлов можно вывести на экран. Макропроцессор **m4** поддерживает до девяти временных файлов (с номерами от 1 до 9). Для перенаправления вывода предназначена встроенная макрокоманда **divert**.

Макрокоманда	Описание
divert (*число***)**	Направляет вывод во временный файл с заданным *номером*.

После того как макропроцессор **m4** встречает в программе функцию **divert**, все выходные данные записываются в конец временного файла с заданным *номером*. Для возобновления вывода на экран вызовите функцию **divert** или **divert(0)**.

По окончании обработки макропроцессор **m4** записывает перенаправленный вывод во временные файлы в соответствии с их номерами. Если вы перенаправили вывод во временный файл, номер которого не входит в интервал от 0 до 9, то макропроцессор **m4** аннулирует вывод.

Для получения данных из временных файлов предназначена встроенная макрокоманда **undivert**.

Макрокоманда	Описание
undivert (*номер-1***, *номер-2...*)**	Добавляет содержимое указанных временных файлов к текущему временному файлу.

Для восстановления выбранных временных файлов в указанном порядке выполните команду **undivert** с аргументами. При выполнении макрокоманды **undivert** макропроцессор **m4** отбрасывает обнаруженные временные файлы и не выполняет в них поиск макроопределений.

Макрокоманда **undivert** не возвращает перенаправленный текст.

Макрокоманда **divnum** позволяет определить временный файл, используемый в данный момент.

Макрокоманда	Описание
divnum	Возвращает номер текущего активного временного файла.

Если вы не изменяете файл вывода с помощью макрокоманды **divert**, то макропроцессор **m4** направляет весь вывод во временный файл с номером 0.

Запуск системных команд из программы

С помощью встроенной макрокоманды **syscmd** вы можете запустить из программы любую команду операционной системы. Например, для выполнения команды **date** необходимо указать следующий оператор:

```
syscmd(date)
```

Создание уникальных имен файлов

Встроенная макрокоманда **maketemp** позволяет сформировать в программе уникальное имя файла.

Макрокоманда	Описание
maketemp (*Строка...nnnnn...Строка***)**	Создает уникальное имя файла, заменяя символы *nnnnn* в строке аргументов на идентификатор текущего процесса.

Например, если указана макрокоманда

```
maketemp(myfilennnnn)
```

то макропроцессор **m4** возвращает строку, состоящую из символов `myfile` и идентификатора процесса. Эту строку можно использовать в качестве имени временного файла.

Работа с условными выражениями

Оценка условных выражений позволяет определять выражения макрокоманд.

Выражение	Описание
ifelse (*строка1***, *строка2***, *аргумент1***, *аргумент2***)**	Если *строка1* совпадает со *строкой2*, возвращается значение *аргумента1*. В противном случае возвращается *аргумент2*.

Встроенная макрокоманда **ifelse** представляет собой условный оператор, выполняющий проверку. В простейшем случае:

```
ifelse(a, b, c, d)
```

сравнивает две строки - a и b.

Если строки a и b одинаковы, стандартная макрокоманда **ifelse** возвращает строку c, если нет - строку d. Например, вы можете задать макроопределение для команды compare, которая будет сравнивать две строки и возвращать слово yes, если они совпадают, и no, если нет:

```
define(compare, `ifelse($1, $2, yes, no)')
```

Кавычки позволяют избежать преждевременной подстановки значений вместо аргументов **ifelse**. Если четвертый аргумент отсутствует, он считается пустым.

В команде **ifelse** может быть любое число аргументов, что позволяет использовать ее вместо громоздкой последовательности условных операторов, реализующих процедуру ветвления. Например:

```
ifelse(a, b, c, d, e, f, g)
```

Эта команда эквивалентна следующей конструкции:

```
if(a == b) x = c;
else if(d == e) x = f;
else x = g;
return(x);
```

Если последний аргумент отсутствует, то результат будет нулевым, поэтому

```
ifelse(a, b, c)
```

возвращает c, если a совпадает с b, и пустое значение в противном случае.

Работа со строками

Макроопределение, приведенное в этом разделе, позволяет преобразовать входные строки в выходные.

Макрокоманда	Описание
len	Возвращает длину строки (аргумента команды) в байтах

Так,

```
len(abcdef)
```

равно 6, а

```
len((a,b))
```

равно 5.

Макрокоманда	Описание
dlen	Возвращает длину графических символов в строке

Символ, которому соответствует двухбайтовый код, на экране выглядит как один символ. Следовательно, результат работы команд **dlen** и **len** для двухбайтовой строки будет различным.

Макрокоманда	Описание
substr (*строка***,** *позиция***,***длина***)**	Возвращает подстроку *строки*, начинающуюся с символа в указанной *позиции* и содержащую заданное *число-символов*.

Команда **substr (***s***,***i***,***n***)** возвращает подстроку строки *s*, которая начинается в *i*-той позиции (позиции нумеруются с нуля), и длина которой составляет *n* символов. Если аргумент *n* отсутствует, возвращается весь остаток строки. Например, функция

```
substr(`now is the time',1)
```

возвращает строку

```
ow is the time
```

Макрокоманда	Описание
index (*строка1***,** *строка2***)**	Возвращает позицию первого символа *строки2* в *строке1* (позиции нумеруются с нуля), либо -1, если *строка1* не содержит *строку2*.

Как и в команде **substr**, нумерация символов строки начинается с нуля.

Макрокоманда	Описание
translit (*строка***,***набор1***,** *набор2***)**	Поиск в *строке* символов из *набора1* и их замена символами из *набора2*.

В общем случае

```
translit(s, f, t)
```

изменяет в `s` все символы из `f` на соответствующие символы из `t`. Например, функция:

```
translit(`little', aeiou, 12345)
```

заменяет гласные соответствующими цифрами и возвращает следующее значение:

```
l3ttl2
```

Если `t` короче, чем `f`, то символы, для которых нет соответствующего символа в `t`, удаляются. Если набор `t` не задан, то символы, входящие в `f`, удаляются из `s`. Так,

```
translit(`little', aeiou)
```

удаляет гласные из строки `little` и возвращает следующее значение:

```
lttl
```

Макрокоманда	Описание
dnl	Удаляет все символы, следующие за этой функцией до символа новой строки, включая сам символ.

Эту макрокоманду можно использовать для удаления пустых строк. Например, функция

```
define(N, 100)
define(M, 200)
define(L, 300)
```

добавляет символ новой строки ко всем строкам, не являющимся частью определения. Символы новой строки передаются на вывод. Для того чтобы избавиться от пустых строк, добавьте после каждого макроопределения встроенную макрокоманду **dnl**:

```
define(N, 100) dnl
define(M, 200) dnl
define(L, 300) dnl
```

Отладка макрокоманд M4

Макрокоманды, описанные в этом разделе, позволяют создать отчет об ошибках, в который также будет включена информация об обработке.

Макрокоманда	Описание
errprint (*строка***)**	Отправляет заданную (*строку*) в стандартный файл вывода сообщений об ошибках.

Пример:

```
errprint (`error')
```

Макрокоманда	Описание
dumpdef (`*имя*'... **)**	Создает дамп текущих имен и определений макрокоманд, указанных в списке *имен*.

Макрокоманда **dumpdef** без аргументов печатает все текущие имена и определения. Не забудьте заключать имена в кавычки.

Дополнительные макрокоманды m4

В таблице перечислены дополнительные макрокоманды **m4** и приведено их краткое описание:

Макрокоманда	Описание
changecom (*l*, *r* **)**	Замена левого и правого символа комментария символами, указанными вместо переменных *l* и *r*.
defn (*имя***)**	Возвращает заключенное в кавычки макроопределение с указанным *именем*
en (*строка***)**	Возвращает число символов в *строке*.
m4exit (*код***)**	Выход из макропроцессора **m4** с указанным *кодом* возврата.
m4wrap (*имя***)**	Выполняет макрокоманду с заданным *именем* при завершении работы макропроцессора **m4**.
popdef (*имя***)**	Замена текущего макроопределения с указанным *именем* предыдущим определением, которое было сохранено с помощью макрокоманды **pushdef**.
pushdef (*имя*, *текст замещения***)**	Сохраняет текущее *макроопределение* с указанным именем и заменяет его на указанный *текст-замещения*.
sysval	Выдает код возврата последней выполненной макрокоманды **syscmd**.

Макрокоманда	Описание
traceoff (*список-макроопределений***)**	Выключает трассировку макроопределений из указанного *списка*. Если *список* не задан, трассировка выключается полностью.
traceon (*имя***)**	Включает трассировку для макроопределения с указанным *именем*. Если *имя* не задано, трассировка включается для всех макроопределений.

Администратор объектных данных

Администратор объектных данных (ODM) предназначен для хранения информации о системе. Эта информация представлена в виде объектов с набором свойств.

ODM может применяться и для работы с данными прикладных программ.

ODM хранит следующую информацию о системе:
- Информацию о конфигурации устройств
- Информацию для интерфейса SMIT (меню, опции и окна диалога)
- Реестров продуктов для процедур установки и обновления
- Информацию о конфигурации линий связи
- Информацию о ресурсах системы

Администратор объектных данных позволяет добавлять, блокировать, сохранять, изменять, считывать, просматривать и удалять объекты и классы объектов. Команды ODM предназначены для выполнения этих задач из командной строки. Функции ODM можно вызывать из приложений.

Некоторые классы объектов поставляются вместе с системой. Эти классы обсуждаются в документации по системным продуктам, к которым они относятся.

Этот раздел содержит следующие подразделы:
- Хранение объектов и классов объектов ODM
- Дескриптор ODM
- Поиск объектов ODM
- Команды и функции ODM
- Пример исходного кода и вывода ODM

Хранение объектов и классов объектов ODM

Основными элементами ODM являются объекты и классы объектов. Для работы с объектами и классами применяются команды и функции ODM. Они позволяют создавать и добавлять классы объектов и объекты для хранения и управления данными.

Термин	Описание
класс объектов	Группа объектов с общим определением. Класс объектов содержит один или несколько дескрипторов. Его структура аналогична таблице. Классы объектов, создаваемые командой **odmcreate** или функцией **odm_create_class**, сохраняются в виде определения массива структур на языке C.
объект	Элемент класса объектов. Предназначен для хранения данных и работы с ними. Он аналогичен логической записи базы данных. Объекты, добавляемые в класс с помощью команды **odmadd** или функции **odm_add_obj**, сохраняются в виде структур языка C в том же файле. Каталог для хранения таких файлов задается при создании класса объектов.

Класс объектов можно рассматривать как массив структур, а объект - как структуру, которая является элементом массива. При добавлении объекта в класс его дескрипторам присваиваются некоторые значения. Для просмотра и изменения значений дескрипторов объектов в ODM предусмотрены специальные функции.

Ниже приведен пример работы с классом объектов и его представителями.

1. Для создания класса объектов Fictional_Characters введите:

```
class Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        char    Friend[20];
};
```

   В данном примере класс объектов Fictional_Characters содержит четыре дескриптора: Story_Star, Birthday и Friend типа char длиной не более 20 символов, а также Age типа short. Для создания файлов ODM, соответствующих описанному классу, необходимо обработать текстовый файл с помощью команды **odmcreate** или функции **odm_create_class**.

2. После создания класса объектов в него можно добавить объекты с помощью команды **odmadd** или функции **odm_add_obj**. Например, вы можете ввести в команде **odmadd** следующий текст для добавления объектов Золушка и Белоснежка в класс Fictional_Characters с указанием значений унаследованных ими дескрипторов:

```
Fictional_Characters:
        Story_Star      = "Золушка"
        Birthday   = "Однажды"
        Age             = 19
        Friend     = "мышь"

Fictional_Characters:
        Story_Star      = "Белоснежка"
        Birthday   = "Однажды"
        Age             = 18
        Friend     = "Фея"
```

   Таблица Fictional_Characters показывает схему класса объектов Fictional_Characters с двумя объектами: Золушка и Белоснежка.

Таблица 74. Fictional Characters

Story Star (char)	Birthday (char)	Age (short)	Friend (char)
Золушка	Однажды	19	Мышь
Белоснежка	Однажды	18	Фея

```
Данные, полученные для 'Story_Star = "Золушка"'
     Золушка:
          Birthday        = Однажды
          Age             = 19
          Friend = Мышь
```

3. После создания класса объектов Fictional_Characters и добавления объектов Золушка и Белоснежка для выражения 'Story_Star = "Золушка"' будут получены следующие сведения:

```
Золушка:
        Birthday        = Однажды
        Age             = 19
        Friend = мышь
```

С помощью команд ODM

При создании или удалении класса объектов с помощью команды **odmcreate** или **odmdrop** нужно указывать каталог с файлом определения класса одним из следующих способов:

1. Для записи файла в каталог по умолчанию, то есть **/etc/objrepos**, укажите **$ODMDIR**.
2. Укажите путь к каталогу в переменной среды **ODMDIR** с помощью команды **export**.

3. Вызовите команду **unset** для удаления значения переменной среды **ODMDIR**, а затем перейдите в каталог, выбранный для хранения классов объектов, с помощью команды **cd**. После этого вызовите команды ODM из этого каталога. Файл, определяющий классы, будет размещен в текущем каталоге.

При вызове команды **odmdelete**, **odmadd**, **odmchange**, **odmshow** или **odmget** для работы с классами и объектами укажите каталог с классами объектов одним из следующих способов:

1. Для работы с классами объектов в каталоге по умолчанию **/etc/objrepos**, укажите переменную **$ODMDIR**
2. Укажите путь к каталогу в переменной среды **ODMDIR** с помощью команды **export**.
3. Сохраните в переменной среды **ODMPATH** список каталогов, разделенных двоеточиями, в которых должен выполняться поиск классов и объектов. Это можно сделать с помощью команды **export**. Например:

   ```
   $ export ODMPATH = /usr/lib/objrepos:/tmp/myrepos
   ```

   Поиск в каталогах из **$ODMPATH** выполняется только в том случае, если в каталоге, указанном в **$ODMDIR**, нет файла определения классов.

Создание классов объектов

Внимание: Изменение файлов, определяющих системные классы и объекты, может привести к появлению неполадок в системе. Обратитесь к администратору перед тем, как использовать каталог **/usr/lib/objrepos** для хранения объектов и классов.

1. Создайте определение одного или нескольких классов объектов в текстовом файле. В разделе “Пример исходного кода и вывода ODM” на стр. 563 приведен пример такого файла с несколькими определениями.
2. Укажите каталог для хранения созданных объектов.

В разделе "Хранение объектов и классов объектов ODM" описаны принципы выбора каталога для хранения объектов и классов при их создании. Большая часть системных объектов и классов объектов хранится в каталоге **/usr/lib/objrepos**.

Создайте пустой класс объектов с помощью команды **odmcreate**, указав текстовый файл с определениями классов объектов в качестве входного файла *ClassDescriptionFile*.

Добавление объектов в класс

Внимание: Изменение файлов, определяющих системные классы и объекты, может привести к появлению неполадок в системе. Обратитесь к администратору перед тем, как использовать каталог **/usr/lib/objrepos** для хранения объектов и классов.

1. Создайте класс, в который будут добавляться объекты. Инструкции по выполнению этой задачи приведены в разделе Создание классов объектов.
2. Создайте определения одного или нескольких объектов. В разделе “Пример исходного кода и вывода ODM” на стр. 563 приведен пример текстового файла, содержащего несколько определений объектов.
3. Укажите каталог для хранения созданных объектов.

   В разделе "Хранение объектов и классов объектов ODM" описаны принципы выбора каталога для хранения объектов и классов при их создании. Большая часть системных объектов и классов объектов хранится в каталоге **/usr/lib/objrepos**.
4. Добавьте объекты в класс с помощью команды **odmadd**, указав текстовый файл с определениями объектов в качестве входного файла *InputFile*.

Блокировка классов объектов

ODM не выполняет автоматической блокировки классов и объектов. За установку и снятие блокировок отвечает приложение, работающее с классами объектов. Для управления блокировкой объектов и классов в

ODM предусмотрены функции **odm_lock** и **odm_unlock**.

Функция	Описание
odm_lock	Обрабатывает строку, которая содержит путь к классу объектов или каталогу классов объектов. Функция возвращает идентификатор блокировки и устанавливает флаг, указывающий, что заданные объекты или классы используются.

После установки флага блокировки функцией **odm_lock** другие процессы могут по-прежнему обращаться к классу объектов. Если существует вероятность конфликта, приложение должно явно проверять наличие флага и ожидать снятия блокировки перед тем, как обратиться к классу объектов.

Другое приложение не может установить блокировку для уже заблокированного объекта или каталога. Однако при блокировке каталога другое приложение может установить блокировку его подкаталога или файлов из этого каталога.

Для разблокирования класса объектов вызовите функцию **odm_unlock**, указав идентификатор блокировки, возвращенный функцией **odm_lock**.

Сохранение объектов и классов

Классы объектов, создаваемые командой **odmcreate** или функцией **odm_create_class**, сохраняются в виде определения массива структур на языке C. Объекты, добавляемые в класс с помощью команды **odmadd** или функции **odm_add_obj**, сохраняются в виде структур языка C в том же файле.

Каталог для хранения таких файлов задается при создании класса объектов.

Способ хранения зависит от способа создания классов и объектов (с помощью команд или функций).

> **Внимание:** Изменение файлов, определяющих системные классы и объекты, может привести к появлению неполадок в системе. Обратитесь к администратору перед тем, как использовать каталог **/usr/lib/objrepos** для хранения объектов и классов.

С помощью функций odm_create_class и odm_add_obj

Функции **odm_create_class** и **odm_add_obj** применяются для создания классов и объектов:

- Если ваше приложение использует для хранения классов объектов каталог, отличный от **ODMDIR**, измените значение этой переменной среды с помощью функции **odm_set_path**. Настоятельно рекомендуется всегда вызывать эту функцию в приложении для настройки каталога хранения при создании классов и объектов.

  ИЛИ
- Перед запуском приложения вызовите команду **set** и задайте необходимый каталог в переменной среды **ODMDIR**.

  ИЛИ
- Сохраните файл в каталоге (**/usr/lib/objrepos**), который содержит большинство системных классов объектов.

Дескрипторы ODM

Дескриптор Администратора объектных данных (ODM) аналогичен переменной, у которой есть тип и имя. При создании класса объектов его дескрипторы определяются как переменные с типами дескрипторов ODM. При добавлении объекта в класс он получает копии всех дескрипторов своего класса. Значения присваиваются объявленным ранее дескрипторам объекта.

ODM поддерживает несколько типов дескрипторов:

Дескриптор	Определение
терминальный дескриптор	Определяет символьный или числовой тип данных.
дескриптор связи	Определяет связи между классами объектов.
дескриптор метода	Определяет операцию или метод объекта.

Дескрипторы объектов и их значения применяются в качестве критериев выбора объектов из класса. Критерий выбора задается в формате, описанном в разделе Поиск объектов ODM. Терминальный дескриптор типа **binary** не может применяться в критериях поиска из-за его неопределенной длины.

Терминальные дескрипторы ODM

Терминальные дескрипторы соответствуют простым типам данных в ODM. Терминальный дескриптор является переменной, имеющий тип терминального дескриптора ODM. Поддерживаются следующие типы терминальных дескрипторов:

Дескриптор	Определение
short	2-байтовое число со знаком.
long	4-байтовое число со знаком.
ulong	Беззнаковое 4-байтовое число.
binary	Строка бит фиксированной длины. Тип binary является пользовательским типом, определенным на этапе создания ODM. Этот тип не может использоваться в критериях поиска.
char	Строка символов фиксированной длины, оканчивающаяся символом NULL.
vchar	Строка символов произвольной длины, оканчивающаяся символом NULL. Терминальный тип дескрипторов **vchar** может применяться в критериях поиска.
long64/ODM_LONG_LONG/int64	8-байтовое число со знаком.
ulong64/ODM_ULONG_LONG/uint64	Беззнаковое 8-байтовое число.

Дескриптор связи ODM

Дескриптор связи ODM определяет связь между объектами из разных классов. Дескриптор связи - это переменная, имеющая тип дескриптора связи ODM.

В следующем фрагменте кода создаются классы объектов **Friend_Table** и **Fictional_Characters**:

```
class Friend_Table {
        char    Friend_of[20];
        char    Friend[20];
};

  class Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        link    Friend_Table Friend_Table Friend_of Friends_of;
};
```

В классе объектов **Fictional_Characters** определен дескриптор связи между дескриптором **Friends_of** и классом объектов `Friend_Table`. При выборке объектов по этой ссылке ODM использует значение дескриптора **Friends_of** и выполняет поиск объектов класса **Friend_Table** с соответствующим значением дескрипторов **Friend_of**. Дескриптор связи класса объектов **Fictional_Characters** определяет класс, на который он ссылается (**Friend_Table**), дескриптор этого класса, с которым нужно создать связь (**Friend_of**), и исходный дескриптор (**Friends_of**) класса **Fictional_Characters**.

Ниже приведен пример определений объектов, которые могут быть добавлены в классы **Fictional_Characters** и **Friend_Table**:

```
Fictional_Characters:
        Story_Star      = "Золушка"
        Birthday   = "Однажды"
        Age             = 19
        Friends_of      = "Золушка"
Fictional_Characters:
        Story_Star      = "Белоснежка"
        Birthday   = "Однажды"
        Age             = 18
        Friends_of      = "Белоснежка"
Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "Фея"
Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "мышь"
Friend_Table:
        Friend_of  = "Белоснежка"
        Friend     = "Ворчун"
Friend_Table:
        Friend_of  = "Белоснежка"
        Friend     = "Соня"
Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "Принц"
Friend_Table:
        Friend_of  = "Белоснежка"
        Friend     = "Счастливчик"
```

Следующие таблицы иллюстрируют схему классов **Fictional_Characters** и **Friend_Table**, их объектов, а также связей между ними.

Story_Star (char)	Birthday (char)	Age (short)	Friends_of (link)
Золушка	Однажды	19	Золушка
Белоснежка	Однажды	18	Белоснежка

```
Данные для 'Story_Star = "Золушка"
     Золушка:
          Birthday        = Однажды
          Age             = 19
          Friends_of      = Золушка
          Friend_of       = Золушка
```

Существует прямая связь между столбцами "**Friends_of**" и "**Friend_of**" этих таблиц. В следующей таблице приведен пример отношения связей двух классов объектов.

Friend_of (char)	Friend (char)
Золушка	Фея
Золушка	Мышь
Белоснежка	Ворчун
Белоснежка	Соня
Золушка	Принц
Белоснежка	Счастливчик

После создания классов **Fictional_Characters** и **Friend_Table** и добавления объектов поиск по критерию `Story_Star = 'Золушка'` даст следующие результаты:

```
Золушка:
        Birthday        = Однажды
        Age             = 19
        Friends_of      = Золушка
        Friend_of       = Золушка
```

Для того чтобы получить дополнительную информацию о связях между классами объектов, вызовите команду **odmget** для класса **Friend_Table**. Для условия `Friend_of = 'Золушка'` будут выданы следующие данные:

```
Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "Фея"

Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "мышь"

Friend_Table:
        Friend_of  = "Золушка"
        Friend     = "Принц"
```

Дескриптор метода ODM

Дескриптор метода ODM позволяет определить класс объектов, содержащий методы или операции. Дескриптор метода - это переменная, имеющая тип дескриптора метода ODM.

Значением дескриптора метода или операции является строка символов, содержащая команду, программу или сценарий оболочки, который запускается при вызове метода. Для каждого объекта класса может быть определен индивидуальный метод. Сами операции не являются частью ODM - они определяются и создаются прикладным программистом.

Для вызова метода, связанного с объектом, предназначена функция **odm_run_method**. Вызов метода является блокирующей операцией - работа ODM приостанавливается до окончания операции.

Например, для создания класса объектов **Supporting_Cast_Ratings** можно задать следующее определение:

```
class Supporting_Cast_Ratings {
        char    Others[20];
        short   Dexterity;
        short   Speed;
        short   Strength;
        method  Do_This;
};
```

В данном примере класс объектов **Supporting_Cast_Ratings** содержит дескриптор метода `Do_This`. Значением дескриптора метода может быть строка, задающая команду, программу или сценарий, вызываемый с помощью функции **odm_run_method**.

Ниже приведен пример добавления объектов в класс **Supporting_Cast_Ratings**:

```
Supporting_Cast_Ratings:
        Friend          = "Соня"
        Dexterity       = 1
        Speed           = 1
        Strength        = 3
        Do_This   = "echo Скорость Сони - 1"

Supporting_Cast_Ratings:
        Others          = "Фея"
        Dexterity       = 10
        Speed           = 10
        Strength        = 10
        Do_This   = "odmget -q "Others='Фея'" Supporting_Cast_Ratings"
```

В следующей таблице приведена схема класса Supporting_Cast_Ratings с дескриптором метода Do_This и операциями, заданными для отдельных объектов этого класса.

Others (char)	Dexterity (short)	Speed (short)	Stength (short)	Do_This (method)
Соня	1	1	3	echo Скорость Сони - 1
Фея	10	10	10	odmget -q "Others='Фея'"Supporting_Cast_Ratings"

Метод **odm_run_method** объекта Соня выдает следующую строку
(с помощью команды **echo**):
"Скорость Сони = 1"

После создания класса Supporting_Cast_Ratings и добавления объектов вызов метода (с помощью функции **odm_run_method**) объекта Соня приведет к тому, что команда **echo** напечатает текст:

```
Скорость Сони = 1
```

Поиск объектов ODM

Во многих функциях ODM требуется выбрать для обработки один или несколько объектов заданного класса. При выборе объектов для определенных функций вы можете указать критерий поиска в форме спецификатора.

qualifier
: В вызове функций ODM - строка, заканчивающаяся символом NULL, определяющая критерий выборки объектов.

Имена дескрипторов и условия выбора, заданные в этом параметре, определяют, какие объекты класса будут выбраны для дальнейшей обработки. Спецификатор содержит один или несколько *предикатов*, объединенных логическими операторами. Каждый предикат состоит из имени дескриптора, оператора сравнения и константы.

Ниже приведен пример спецификатора, содержащего три предиката, объединенных двумя логическими операторами:

```
SUPPNO=30 AND (PARTNO>0 AND PARTNO<101)
```

В этом примере спецификатором является вся строка. Три предиката SUPPNO=30, PARTNO>0 и PARTNO<101 связаны логическим оператором AND. В первом предикате SUPPNO - имя дескриптора, = (знак равенства) - это оператор сравнения, а 30 - константа, с которой сравнивается значение дескриптора.

Каждый предикат задает ограничение на значения дескрипторов объектов класса. Выбираются все объекты, дескрипторы которых удовлетворяют указанному ограничению. Первый предикат указывает, что должны быть выбраны все объекты, у которых дескриптор (SUPPNO) равен (=) константе (30).

Часть спецификатора в скобках

```
PARTNO>0 AND PARTNO<101
```

содержит два предиката, объединенных логической операцией AND (И). Эти предикаты указывают, что значение дескриптора PARTNO должно быть больше 0, но меньше 101. Это условие построено из двух предикатов, объединенных оператором И. Например, если дескриптор PARTNO обозначает номер детали в реестре компании, то вторая часть спецификатора определяет набор деталей с номерами от 0 до 101.

В другом примере спецификатор

```
lname='Smith' AND Company.Dept='099' AND Salary<2500
```

позволяет выбрать всех служащих (все объекты ODM) с фамилией Иванов, работающих в отделе 099 и получающих зарплату менее 2500 долл. Обратите внимание на то, что имя дескриптора Dept указано со спецификатором класса Company для создания уникального идентификатора.

Имена дескрипторов в предикатах ODM

В ODM имя дескриптора не обязательно должно быть уникальным. Одно имя дескриптора может использоваться в нескольких классах. В этом случае для того, чтобы однозначно идентифицировать дескриптор, вместе с его именем указывается имя класса.

Операторы сравнения в предикатах ODM

Ниже приведен список допустимых операторов сравнения:

Оператор	Определение
=	Равно
!=	Не равно
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно
LIKE	Поиск строк по шаблону

Сравниваться могут только совместимые типы данных.

Оператор сравнения LIKE

Операция LIKE позволяет найти дескрипторы типа char, соответствующие заданному шаблону. Например, предикат

```
NAME LIKE 'ANNE'
```

задает поиск значения ANNE в дескрипторах NAME всех объектов класса. Это выражение эквивалентно

```
NAME = 'ANNE'
```

Оператор LIKE поддерживает следующие символы подстановки:

- Символ ? (вопросительный знак) обозначает один символ. Предикат

  ```
  NAME LIKE '?A?'
  ```

  задает критерий поиска трехбуквенных строк с символом *A* посередине в дескрипторе NAME объекта. Этому критерию удовлетворяют значения PAM, DAN и PAT.
- Символ * (звездочка) обозначает произвольную строку из нуля или более символов. Предикат

  ```
  NAME LIKE '*ANNE*'
  ```

  задает поиск значений, включающих текст ANNE, в дескрипторах NAME всех объектов класса. Этому критерию удовлетворяют значения LIZANNE, ANNETTE и ANNE.
- Символы [ ] (квадратные скобки) обозначают один из символов, заключенных в скобки. Предикат

  ```
  NAME LIKE '[ST]*'
  ```

  задает поиск значений, начинающихся с символа *S* или *T* в дескрипторе NAME.

  Символ - (знак минус) позволяет задать диапазон значений. Предикат

  ```
  NAME LIKE '[AD-GST]*'
  ```

  задает поиск значений, начинающихся с символа *A, D, E, F, G, S* или *T*.
- Символы [!] (восклицательный знак в квадратных скобках) соответствуют любому символу, кроме указанных в скобках. Предикат

  ```
  NAME LIKE '[!ST]*'
  ```

  задает поиск всех значений, кроме начинающихся с символа *S* или *T*, в дескрипторе NAME.

Критерий поиска может содержать любую комбинацию символов подстановки.

Константы в предикатах ODM

В предикате ODM можно указывать числовые и строковые константы:

1. Числовые константы в предикатах ODM представляют собой число (с десятичной точкой или без), перед которым может стоять знак минус, а после - символ экспоненциальной записи *E* или *e*. Если указан символ *E* или *e*, после него должно стоять значение порядка, которое также может иметь знак.

   Ниже приведены примеры допустимых числовых констант:

   ```
   2            2.545   0.5   -2e5   2.11E0
   +4.555e-10   4E0     -10    999   +42
   ```

   Значение E0 указывает отсутствие экспоненты.

2.

   Строковые константы в предикатах ODM должны быть заключены в одиночные кавычки:

   ```
   'smith'   '91'
   ```

   Строковые константы могут быть произвольной длины. Одинарные кавычки внутри строки символов нужно удваивать. Пример:

   ```
   'DON''T GO'
   ```

   обозначает строку

   ```
   DON'T GO
   ```

Логический оператор AND в предикатах

В предикатах ODM может применяться логический оператор AND (И). Он может быть задан в форме AND или and.

Логический оператор AND может соединять несколько предикатов. Например, спецификатор

```
predicate1 AND predicate2 AND predicate3
```

обозначает `предикат-1`, объединенный с `предикатом-2`, а затем - с `предикатом-3`.

Команды и функции ODM

Администратор объектных данных позволяет добавлять, блокировать, сохранять, изменять, считывать, просматривать и удалять объекты и классы объектов. Команды ODM вводятся в командной строке.

Для работы с объектами и классами в программе на языке C могут применяться функции ODM. Если функция ODM завершается неудачно, она возвращает значение -1. Диагностическая информация об ошибке передается через внешнюю переменную **odmerrno** (определенную в файле **odmi.h**). Коды ошибок ODM также описаны в файле **odmi.h**.

> **Примечание:** Если в программе на языке C применяются данные функции, укажите следующую опцию: **-binitfini:__odm_initfini_init:__odm_initfini_fini**.

Команды

Ниже перечислены команды ODM:

Команда	Описание
odmadd	Добавляет объекты в класс. Команда **odmadd** получает на входе текстовый файл настройки и добавляет в классы объекты, описанные в этом файле.
odmchange	Изменяет указанные объекты в заданном классе.
odmcreate	Создает пустые классы объектов. Команда **odmcreate** получает на входе текстовый файл, описывающий классы объектов, и создает файлы **.h** и **.c** для работы с объектами этих классов в приложениях на языке C.
odmdelete	Удаляет объекты из класса.
odmdrop	Удаляет класс объектов.
odmshow	Выводит описание класса объектов. Команда **odmshow** получает на входе имя класса объектов и выводит информацию о классе в формате команды **odmcreate**.
odmget	Получает объекты из класса и выводит информацию об объектах в формате команды **odmadd**.

Функции

Ниже перечислены функции ODM:

Функция	Описание
odm_add_obj	Добавляет новый объект в класс.
odm_change_obj	Изменяет содержимое объекта.
odm_close_class	Закрывает класс объектов.
odm_create_class	Создает пустой класс объектов.
odm_err_msg	Возвращает сообщение об ошибке.
odm_free_list	Освобождает память, выделенную для функции **odm_get_list**.
odm_get_by_id	Возвращает объект с заданным идентификатором.
odm_get_first	Возвращает первый объект из класса, удовлетворяющий критерию поиска.
odm_get_list	Возвращает список объектов класса, удовлетворяющих критерию поиска.
odm_get_next	Возвращает следующий объект класса, удовлетворяющий критерию поиска.
odm_get_obj	Возвращает объект класса, удовлетворяющий заданному критерию.
odm_initialize	Инициализирует сеанс ODM.
odm_lock	Блокирует класс или группу классов.
odm_mount_class	Возвращает символьную структуру указанного класса объектов.
odm_open_class	Открывает класс объектов.
odm_rm_by_id	Удаляет объект с заданным идентификатором.
odm_rm_obj	Удаляет все объекты из класса, удовлетворяющие заданному критерию.
odm_run_method	Вызывает метод указанного объекта.
odm_rm_class	Удаляет класс объектов.
odm_set_path	Задает каталог по умолчанию для хранения классов объектов.
odm_unlock	Разблокирует класс или группу классов.
odm_terminate	Завершает сеанс ODM.

Пример исходного кода и вывода ODM

На диаграмме Классы объектов Fictional_Characters, Friend_Table и Enemy_Table показаны классы и объекты, создаваемые в примерах из этого раздела.

Таблица 75. Fictional_Characters

Story_Star (char)	Birthday (char)	Age (short)	Friends_of (link)	Enemies_of (link)	Do_This (method)
Золушка	Однажды	19	Золушка	Золушка	echo Наряжается
Белоснежка	Однажды	18	Белоснежка	Белоснежка	echo Наряжается

Таблица 76. Friend_Table

Friend_of (char)	Friend (char)
Золушка	Фея
Золушка	Мышь
Белоснежка	Ворчун
Белоснежка	Соня
Золушка	Принц
Белоснежка	Счастливчик

Таблица 77. Enemy_Table

Enemy_of (char)	Enemy (char)
Золушка	Полночь
Золушка	Мачеха
Белоснежка	Мачеха

Пример исходного кода для создания классов объектов ODM

Ниже приведен пример файла **MyObjects.cre**, служащего для создания трех классов объектов с помощью команды **odmcreate**:

```
*      MyObjects.cre
*       Входной файл утилиты создания классов объектов ODM.
*       Создает три класса объектов:
*               Friend_Table
*               Enemy_Table
*               Fictional_Characters

class Friend_Table {
        char    Friend_of[20];
        char    Friend[20];
};

class   Enemy_Table {
        char    Enemy_of[20];
        char    Enemy[20];
};

class Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        link    Friend_Table Friend_Table Friend_of Friends_of;
        link    Enemy_Table Enemy_Table Enemy_of Enemies_of;
        method  Do_This;
};

* Конец входного файла MyObjects.cre для утилиты создания классов ODM. *
```

Класс объектов `Fictional_Characters` содержит шесть дескрипторов:

- `Story_Star` и `Birthday` типа char длиной до 20 символов.
- `Age` типа short.
- Дескрипторы связей `Friends_of` и `Enemies_of`, ссылающиеся на два определенных ранее класса объектов.

  Примечание: Ссылка на класс объектов повторяется дважды.
- `Do_This` является дескриптором метода.

Для создания файлов ODM, соответствующих описанному классу, необходимо обработать текстовый файл с помощью команды **odmcreate**

Пример вывода ODM для определений классов объектов

В результате обработки файла **MyObjects.cre** командой **odmcreate** создается файл **.h** со следующими структурами:

```
* MyObjects.h
* Вывод ODM после обработки входного файла MyObjects.cre.
* Определяет структуры для трех классов:
*               Friend_Table
*               Enemy_Table
*       Fictional_Characters
#include <odmi.h>

struct  Friend_Table {
       long     _id;        * уникальный ИД объекта в классе *
       long     _reserved;  * зарезервированное поле *
       long     _scratch;   * дополнительное поле для использования приложением *
       char    Friend_of[20];
       char    Friend[20];
};

#define Friend_Table_Descs 2
extern struct Class Friend_Table_CLASS[];
#define get_Friend_Table_list(a,b,c,d,e) (struct Friend_Table * )odm_get_list (a,b,c,d,e)

struct  Enemy_Table {
        long    _id;
        long    _reserved;
        long    _scratch;
        char    Enemy_of[20];
        char    Enemy[20];
};
#define Enemy_Table_Descs 2
extern struct Class Enemy_Table_CLASS[];
#define get_Enemy_Table_list(a,b,c,d,e) (struct Enemy_Table * )odm_get_list (a,b,c,d,e)

struct  Fictional_Characters {
        long    _id;
        long    _reserved;
        long    _scratch;
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
       struct  Friend_Table *Friends_of;    * связь *
        struct  listinfo *Friends_of_info;   * связь *
        char    Friends_of_Lvalue[20];       * связь *
        struct  Enemy_Table *Enemies_of;     * связь *
        struct  listinfo *Enemies_of_info;   * связь *
        char    Enemies_of_Lvalue[20];       * связь *
        char    Do_This[256];                * метод *
};

#define Fictional_Characters_Descs 6

extern struct Class Fictional_Characters_CLASS[];
#define get_Fictional_Characters_list(a,b,c,d,e)
(struct Fictional_Characters * )odm_get_list (a,b,c,d,e)

* Конец файла MyObjects.h с описанием выходных структур после обработки ODM    *
```

Пример исходного кода для добавления объектов в класс ODM

Ниже приведен пример определений, после обработки которых командой **odmadd** соответствующие объекты будут добавлены в класс, созданный в результате обработки файла **MyObjects.cre**.

```
* MyObjects.add
*       Входной файл утилиты создания объектов ODM.
*       Добавляет объекты в классы:
*               Friend_Table
*               Enemy_Table
*               Fictional_Characters

Fictional_Characters:
Story_Star = "Золушка" # комментарий файла MyObjects.add
Birthday   = "Однажды"
Age             = 19
Friends_of      = "Золушка"
Enemies_of = "Золушка"
Do_This    = "echo Убирается в доме"

Fictional_Characters:
Story_Star = "Белоснежка"
Birthday   = "Однажды"
Age        = 18
Friends_of = "Белоснежка"
Enemies_of = "Белоснежка"
Do_This   =              "echo Наряжается"

Friend_Table:
Friend_of  = "Золушка"
Friend     = "Фея"

Friend_Table:
Friend_of  = "Золушка"
Friend     = "мышь"

Friend_Table:
Friend_of  = "Белоснежка"
Friend     = "Ворчун"

Friend_Table:
Friend_of  = "Белоснежка"
Friend     = "Соня"

Friend_Table:
Friend_of  = "Золушка"
Friend     = "Принц"

Friend_Table:
Friend_of  = "Белоснежка"
Friend     = "Счастливчик"

Enemy_Table:
Enemy_of   = "Золушка"
Enemy      = "полночь"

Enemy_Table:
Enemy_of   = "Золушка"
Enemy      = "Злая мачеха"

Enemy_Table:
Enemy_of   = "Белоснежка"
Enemy      = "Злая мачеха"
* Конец входного файла MyObjects.add для утилиты добавления объектов ODM. *
```

Примечание: комментарии, указанные в предыдущем примере с помощью символов ***** **#**, в файл объекта записаны не будут. Если в начале строки расположен символ комментария, команда не добавляется в файл объекта. Комментарий будет добавлен в файл только в том случае, если он заключен в двойные кавычки (" ").

Одновременное выполнение нитей

Одновременное выполнение нитей - это технология, позволяющая физическому процессору одновременно обрабатывать несколько инструкций из контекстов нескольких аппаратных нитей. Поскольку на каждый физический процессор приходятся по две аппаратные нити, несколько инструкций могут выполняться одновременно.

Одновременное выполнение нитей позволяет воспользоваться всеми преимуществами суперскалярной архитектуры процессора - одновременно выполнять два приложения на одном и том же процессоре. Ни одно приложение не может занять все ресурсы процессора.

Достоинства Одновременное выполнение нитей

Он дает максимальный эффект в коммерческих средах, в которых общее количество выполняемых операций играет более важную роль, чем скорость выполнения отдельной операции. Одновременное выполнение нитей позволяет повысить пропускную способность систем с большой нагрузкой, часто меняющейся по интенсивности, например серверов баз данных и Web-серверов.

Максимальный эффект одновременное выполнение нитей достигается на приложениях с высоким числом циклов на инструкцию (CPI). Для таких приложений характерна неоптимальная загрузка процессора и ресурсов памяти. Высокий показатель CPI обычно бывает вызван частыми промахами при обращении к кэшу в ходе выполнения объемных приложений. Эффективность крупномасштабных коммерческих сред в определенной степени зависит от того, связаны ли аппаратные нити, или же они выполняются полностью независимо друг от друга. Такая зависимость характерна прежде всего для крупномасштабных коммерческих сред. Если массовые задания, образующие основную нагрузку на систему, пользуются одними и теми же инструкциями или данными, то одновременное выполнение нитей может существенно повысить производительность.

одновременное выполнение нитей не очень эффективно в средах, в которых большинство приложений используют много независимых друг от друга процессорных ресурсов и ресурсов памяти. Типичным примером ситуации, когда одновременное выполнение нитей может привести даже к снижению производительности, может служить среда с большим объемом вычислений с плавающей точкой. В такой среде интенсивно используются ресурсы блока вычислений с плавающей точкой и память. В средах с низким коэффициентом CPI и низким процентом промахов при обращении к кэшу может достигаться небольшой выигрыш.

Измерения, проведенные на выделенном разделе с коммерческой нагрузкой, продемонстрировали повышение пропускной способности на 25%-40%. Одновременное выполнение нитей должно обеспечивать более рациональную загрузку процессоров в общих разделах. Дополнительные нити позволяют резко повысить производительность раздела после включенияодновременное выполнение нитей, поскольку на восстановление рабочего набора будет требоваться меньше времени. Как следствие, нити будет выполняться так же, как если бы они выполнялись в выделенном разделе. Хотя это на первый взгляд неочевидно, одновременное выполнение нитей показывает себя лучше всего в ситуациях, когда кэш работает с минимальной эффективностью.

Установка режима с помощью команды smtctl

В AIX можно управлять режимом раздела для одновременное выполнение нитей с помощью команды **smtctl**. Эта команда позволяет включать и отключать одновременное выполнение нитей в масштабах всей системы как немедленно, так и при следующей загрузке. Текущий режим одновременное выполнение нитей сохраняется при перезагрузке системы. По умолчанию AIX включает одновременное выполнение нитей.

Формат команды **smtctl** следующий:

```
smtctl [ -m { off  | on } [ { -boot | -now } ] ]
```

Дополнительная информация приведена в описании команды **smtctl** в книге *Справочник по командам, том 5*.

Консоль аппаратного обеспечения Конфигурация Одновременное выполнение нитей

При настройке разделов с общими процессорами в консоли HMC нужно указать минимальное, предпочитаемое и максимальное число виртуальных процессоров. Для выделенных разделов действуют аналогичные параметры, но в другой терминологии. При работе с выделенными разделами процессоры всегда называются процессорами.

В обеих моделях разбиения системы на разделы нужно указать параметры предоставления процессоров разделам в момент загрузки и в дальнейшем в ходе работы раздела. Если это возможно, при запуске системы разделу предоставляется указанное в его конфигурации предпочитаемое число процессоров. Если это невозможно, POWER Hypervisor выделит меньше процессоров, но не менее минимума, указанного в конфигурации раздела.

От количества процессоров, указанных в HMC, зависит количество логических процессоров, предоставляемых операционной системой AIX. Если раздел поддерживает одновременное выполнение нитей, количество предоставленных AIX логических процессоров будет вдвое больше максимума, указанного в конфигурации, поскольку на каждый процессор приходятся по две аппаратные нити, а AIX рассматривает каждую аппаратную нить как отдельный логический процессор. Это позволяет AIX включать и отключать одновременное выполнение нитей без перезагрузки раздела.

Динамические логические разделы для Одновременное выполнение нитей

Средства Динамические логические разделы (DLPAR) консоли HMC позволяют изменять количество процессоров, предоставленных работающим разделам. Можно добавлять и удалять процессоры в пределах ограничений, указанных в конфигурации раздела. При добавлении процессора в раздел, поддерживающий одновременное выполнение нитей, AIX запускает обе аппаратные нити и активирует два логических процессора. При удалении процессора из раздела, поддерживающего одновременное выполнение нитей, AIX останавливает обе аппаратные нити и отключает два логических процессора.

В режиме одновременное выполнение нитей генерируются два события DLPAR. Каждому добавляемому или удаляемому логическому процессору соответствует отдельное событие. API, используемые в сценариях DLPAR, оперируют логическими разделами, поэтому количество событий DLPAR зависит от количества добавляемых и удаляемым логических процессоров. Если в разделе не включено одновременное выполнение нитей, то генерируется только одно событие DLPAR. AIX автоматически преобразует запрос DLPAR, поступивший от консоли HMC, в соответствующий набор событий DLPAR.

Micro-Partitioning и Одновременное выполнение нитей

POWER Hypervisor™ сохраняет и восстанавливает данные о состоянии процессоров при распределении и замещении виртуальных процессоров. Для процессоров, поддерживающих одновременное выполнение нитей, это данные о двух активных контекстах нитей. Каждая аппаратная нить в системе AIX рассматривается как отдельный логический процессор. Поэтому если выделенному разделу отведен один физический процессор, в AIX он будет настроен как два логических процессора. Поскольку это правило действует для всех типов разделов, то общий раздел с двумя виртуальными процессорами будет настроен в AIX как раздел с четырьмя логическими процессорами, а общий раздел с четырьмя виртуальными процессорами будет настроен в AIX как раздел с восемью логическими процессорами. Спаренные нити всегда предоставляются одному разделу.

Общие процессорные ресурсы всегда предоставляются таким образом, чтобы физический процессор был полностью предоставлен одному разделу. Без одновременное выполнение нитей, в AIX раздел с 4 виртуальными процессорами и правами на 200 процессорных ресурсов настраивается как раздел с 4 логическими процессорами, в котором каждый логический процессор обладает мощностью в 50% от мощности физического процессора. При наличии одновременное выполнение нитей, вместо раздела с 4 логическими процессорами появляется раздел с 8 логическими процессорами, в котором мощность каждого логического процессора составляет около 25% от мощности физического процессора. Однако в среде с одновременное выполнение нитей не существует прямой линейной зависимости между скоростью выполнения нитей и относительным объемом ресурсов процессора. Поскольку обе аппаратные нити предоставляются разделу вместе, они активны на протяжении 50% окна предоставления, и в силу того что они выполняются на одном процессоре, логически это составляет около 25% от ресурсов процессора. Таким образом, каждый логический процессор может обрабатывать прерывания в течение вдвое более длительного времени, чем это возможно для каждого из них по отдельности.

Приоритеты аппаратных нитей

Процессор поддерживает возможность установки приоритета для аппаратных нитей. От разницы в приоритетах нитей зависит процентное распределение физических ячеек декодирования между ними. Чем больше ячеек предоставлено нити, тем выше скорость ее выполнения. По умолчанию AIX устанавливает равный приоритет для нитей, однако в определенных обстоятельствах меняет его для оптимизации производительности. Например, AIX снижает приоритет нити на время, пока она выполняет пустой цикл или ждет блокировки ядра. Приоритет нити повышается на время, пока она удерживает полную блокировку ядра. Эти операции с приоритетами не видны на уровне пользователя. AIX не учитывает приоритеты программных нитей при изменении приоритета аппаратных нитей.

Нагрузка в первую очередь распределяется между основными нитями, и только после этого - между вспомогательными. Производительность нити выше всего тогда, когда парная к ней нить простаивает. Параметры привязки нитей также используются при оптимизации простоев и рутинной оптимизации очереди выполнения.

Динамическое распределение ресурсов

Разбиение системы на логические разделы аналогично организации разделов на жестком диске. Один единственный физический жесткий диск разбивается на разделы так, что операционная система воспринимает его как несколько отдельных логических дисков.

На каждом разделе можно установить операционную систему, и работать с ним как с отдельной физической системой.

Логические разделы (LPAR) - это разделение процессоров, памяти и аппаратных ресурсов на несколько сред, так что каждая среда может управляться отдельно своей собственной операционной системой и приложениями. Допустимое число логических разделов зависит от системы. Обычно разделы применяются для различных целей: для работы с базами данных, средой клиент/сервер, Web-сервером, тестовой и рабочей средой и т.д. Разделы взаимодействуют между собой точно так же, как отдельные системы.

Динамическое распределение ресурсов между разделами (DLPAR) позволяют логически добавлять и удалять ресурсы компьютера к логическому разделу, не перезагружая операционную систему. Ниже приведены некоторые функции, поддерживаемые DLPAR:

- Функция *Модернизация по запросу* (*CUoD*) IBM System p позволяет вводить в действие установленные, но еще не активные процессоры при изменении требований к ресурсам.
- Функция динамическое отключение процессоров реализована на серверах IBM Power и в некоторых моделях SMP. динамическое отключение процессоров позволяет динамически выключать процессор при достижении внутреннего порога количества ошибок. В сочетании с DLPAR, функция динамическое отключение процессоров позволяет динамически заменять поврежденный процессор на один из резервных. Такая замена не влияет на приложения и расширения ядра.
- DLPAR позволяет управлять загрузкой нескольких разделов, что особенно важно при работе с большими серверами, позволяющими перемещать ресурсы между разделами.

Запросы DLPAR построены на основе простых запросов на удаление и добавление ресурсов в логические разделы. Пользователь может выполнять эти команды так же, как и запросы на перемещение ресурсов в Консоль аппаратного обеспечения (HMC), управляющей всеми операциями DLPAR. Операции DLPAR выполняются на уровне встроенного программного обеспечения System p и AIX.

Информация, связанная с данной:

cpupstat

drmgr

dr_reconfig

reconfig

Программы, поддерживающие и допускающие DLPAR

Программа, допускающая DLPAR, не прерывается аварийно при выполнении операций DLPAR.

Производительность программы может быть снижена в результате удаления ресурсов или остаться неизменной несмотря на их добавление, однако программа по-прежнему будет работать. Кроме того, программа, допускающая DLPAR, может отменить выполнение операции DLPAR, если у нее есть зависимости, о которых известно операционной системе.

Программа, поддерживающая *DLPAR*, с помощью кода DLPAR самостоятельно отслеживает и обрабатывает изменения конфигурации ресурсов. Это может быть реализовано следующими способами:

- Путем регулярного опроса системы на предмет изменений в ее конфигурации.
- Путем регистрации специального кода, который автоматически уведомляется при изменении конфигурации системы.

Программы, поддерживающие *DLPAR*, должны быть разработаны таким образом, чтобы при выполнении операций DLPAR как минимум не возникали ошибки. Рекомендуется, чтобы программы, поддерживающие DLPAR, также самостоятельно управляли производительностью и масштабируемостью. Это значительно более сложная задача, поскольку при удалении или добавлении памяти может потребоваться очистить и изменить размер буферов. Кроме того, при изменении числа включенных процессоров необходимо динамически изменить число нитей. Число нитей зависит не только от числа процессоров. Например, самый лучший способ снизить объем памяти, занимаемой программами на Java, - это уменьшить число нитей, поскольку это сокращает число активных объектов, которые должен обрабатывать сборщик мусора виртуальной машины Java.

Большинство приложений по умолчанию допускают DLPAR.

Создание программ, допускающих DLPAR

DLPAR может вызвать следующие типы ошибок двоичной несовместимости:

Примечание: Эти ошибки вызваны добавлением процессора.

- Если программа была оптимизирована для систем с одним процессором, но число процессоров в разделе увеличилось во время выполнения операции проверки, то может возникнуть ошибка. Ошибки также могут возникнуть в программах, содержащих операции блокировки для однопроцессорной системы, без инструкций sync и isync. Эти инструкции обязательны в коде, изменяющем самого себя, а также в сгенерированном коде, и, следовательно в системах с DLPAR. Постарайтесь найти все программы, в которых предполагается, что в системе есть только один процессор. В них необходимо включить функции, определяющие число активных процессоров.

  Программы могут определить число включенных процессоров следующим образом:

  – Путем загрузки поля *_system_configuration.ncpus*
  – *var.v_ncpus*
  – С помощью системного вызова **sysconf** с флагом **_SC_NPROCESSORS_ONLN**.
- Программы, индексирующие данные по номеру процессора, используют для определения этого номера системный вызов **mycpu**. При добавлении нового процессора это может вызвать ошибку, поскольку пути к данным могут не быть правильно инициализированы и распределены. Сбой произойдет в программах, создающих список процессоров заранее, так как число процессоров может изменяться с DLPAR динамически.

  Чтобы избежать этой ошибки, индексируйте данные по максимальному числу одновременно включенных процессоров. Значение *N* числа процессоров, которые могут поддерживаться операционной системой, предпочтительнее значения *N* включенных в данный момент процессоров. Максимальное число процессоров постоянно, в то время как число включенных процессоров увеличивается и уменьшается при включении и выключении процессоров. Минимальное и максимальное число процессоров задаются при создании раздела. Максимальное значение отражено в следующих переменных:

- *_system_configuration.max_ncpus*
- *_system_configuration.original_ncpus*
- *var.v_ncpus_cfg*
- **sysconf** (**_SC_NPROCESSORS_CONF**)

Переменные *_system_configuration.original_ncpus* и *var.v_ncpus_cfg* предопределены заранее. В системах с DLPAR они указывают максимально возможное число процессоров. В системах без DLPAR значение этих переменных отражает число процессоров, настроенное при загрузке. Обе эти переменные отражают максимальное поддерживаемое значение, вне зависимости от операций, выполняемых динамическое отключение процессоров. Эти предопределенные переменные рекомендуется использовать в приложениях, созданных в AIX 4.3, поскольку это гарантирует применение одного и того же кода в AIX 4.3 и более поздних версиях. Если приложение должно инициализировать зависящие от процессоров данные динамически, то зарегистрируйте обработчик DLPAR, который будет вызываться перед добавлением процессора.

Создание программ, поддерживающих DLPAR

Программа, поддерживающая *DLPAR*, может самостоятельно распознавать изменения конфигурации системы и динамически адаптироваться к ним. Код такой программы может не соответствовать модели DLPAR, например, в него может быть встроен системный монитор, позволяющий определить изменения в конфигурации системы. Такой подход позволяет добиться ограниченного повышения производительности, но неприменим для эффективной адаптации к крупным изменениям в системе, поскольку интегрирует программу с DLPAR. Например, системный монитор не подходит для компьютеров с оперативным подключением процессоров, так как одновременно может быть подключено несколько процессоров и карт памяти. Он также неприменим для работы с зависимостями приложений, такими как связывание с процессором, поскольку некоторые действия должны быть предприняты до непосредственного удаления процессора DLPAR.

Технологии DLPAR могут применяться в приложениях следующих типов:

- Приложения, логика работы которых зависит от конфигурации системы, включая следующие:
  - Определяющие число включенных процессоров и объем памяти при запуске приложения.
  - Отрабатывающие внешние инструкции на основе конфигурации процессоров и памяти, что позволяет использоваться максимально возможное число нитей, максимальный объем буферов и закрепленной памяти.
- Приложения, которые сами определяют число включенных процессоров и объем памяти, включая следующие:
  - Мониторы производительности
  - Средства отладки
  - Программы устранения ошибок
  - Администраторы загрузки
  - Администраторы лицензий

    Примечание: DLPAR применяется не во всех администраторах лицензий, особенно при лицензировании числа пользователей.
- Приложения, закрепляющие данные, текст или стек приложений с помощью системного вызова **plock**.
- Приложения, использующие общие сегменты памяти System с помощью **PinvOption** (**SHM_PIN**)
- Приложения, связывающие нити с процессорами с помощью системного вызова **bindprocessor**.

Динамическое распределение ресурсов между разделами для больших страниц памяти не поддерживаются. Объем памяти, заранее выделенной пулу больших страниц, может повлиять на способность DLPAR работать с памятью. Область памяти с большими страницами удалить нельзя. Разработчики программ должны предоставить возможность не использовать большие страницы.

Создание программ, поддерживающих DLPAR, с помощью API DLPAR

Для поддержки DLPAR предназначены специальные интерфейсы программирования. На разных этапах работы с динамическое распределение ресурсов между разделами приложениям отправляется сигнал **SIGRECONFIG**. Обычная операция с подсистемой DLPAR состоит из этапа проверки, предварительного и завершающего этапа. Получив этот сигнал, приложения могут с помощью системных вызовов DLPAR узнать, какая операция выполняется, и предпринять необходимые действия.

Примечание: Если приложения не будут блокировать сигналы, то загрузка системы может привести к нарушению синхронизации нитей. Приложение должно ждать в течение короткого периода времени, а затем переходить к следующему этапу. Ждать неограниченное время не рекомендуется, вся операция DLPAR может быть отменена из-за зависшей непривилегированной нити.

Для правильной доставки сигналов приложение может использовать маску сигналов и приоритет планирования. Код, поддерживающий DLPAR, может быть напрямую встроен в алгоритм. Кроме того, обработчик сигналов может состоять из модулей, находящихся в разных подключаемых библиотеках.

Для обработки события DLPAR с помощью API выполните следующие действия:

1. Отслеживайте сигнал **SIGRECONFIG** с помощью системного вызова **sigaction**. По умолчанию этот сигнал игнорируется.
2. Контролируйте маску сигнала как минимум в одной нити, чтобы сигнал доставлялся в реальном времени.
3. Убедитесь, что приоритет нити, получающий сигнал, достаточен для быстрой обработки полученного сигнала.
4. С помощью системного вызова **dr_reconfig** узнайте тип ресурса, тип действия, этап события и другую необходимую информацию.

   Примечание: Системный вызов **dr_reconfig** применяется внутри обработчика для определения цели запроса DLPAR.

Управление зависимостями DLPAR приложения

Запрос на удаление DLPAR может вызвать ошибку по нескольким причинам. Наиболее распространенная из них - это занятость ресурса или нехватка системных ресурсов для выполнения запроса.

В этих случаях ресурс остается в обычном состоянии, как если бы событие DLPAR никогда не происходило.

Основная причина сбоя при *удалении процессора* - это связывание процессора. Сбой возникает, если операционная система не может обрабатывать связывания процессора или операции DLPAR, либо приложения не могут продолжать нормальную работу. Для исправления ошибки отмените связывание, установите новую связь или завершите приложение. При этом затрагиваются процессы и нити для используемого типа связывания.

Основной причиной сбоя *при удалении памяти* является нехватка в системе закрепленной памяти. Эта ошибка затрагивает всю систему и может не быть вызвана конкретным приложением. Если в удаляемой области есть закрепленная страница, то ее содержимое должно быть перенесено в другую страницу с соответствующей коррекцией преобразования физических адресов в виртуальные. Если для перемещения страницы недостаточно памяти, то произойдет ошибка. Для того чтобы снизить вероятность ее возникновения, уменьшите объем закрепленных страниц в системе. Для этого можно уничтожить закрепленные сегменты памяти, прервать программы с системными вызовами **plock** или удалить **plock** из программы.

Как правило, ошибки при *удалении разъема PCI* происходят в случае, если адаптер в разъеме занят. Зависимости устройств не отслеживаются. Например, разъем может зависеть от адаптера, устройства, группы томов, логического тома, файловой системы или файла. В таком случае удалите зависимость, остановив приложения, размонтировав файловую систему или выключив группу томов.

Понятия, связанные с данным:

“Связывание процессора”
Приложения можно связать с процессорами с помощью системного вызова **bindprocessor**. При этом предполагается, что нумерация процессоров начинается с нуля (0), и заканчивается на *N*-1, где *N* - число включенных CPU.

Связывание процессора

Приложения можно связать с процессорами с помощью системного вызова **bindprocessor**. При этом предполагается, что нумерация процессоров начинается с нуля (0), и заканчивается на *N*-1, где *N* - число включенных CPU.

N определяется программой путем чтения системной переменной **_system_configuration.ncpus** . При добавлении и удалении процессоров значение этой переменной увеличивается и уменьшается с помощью динамическое распределение ресурсов между разделами.

Учтите, что система нумерации непрерывная. Процессоры всегда добавляются в позицию *N* и удаляются из позиции *N*-1. Систему нумерации **bindprocessor** нельзя применять для связывания конкретного логического процессора, поскольку может быть удален любой процессор, что не будет отражено в нумерации (всегда удаляется процессор *N*1). По этой причине, идентификаторы, используемые системным вызовом **bindprocessor**, называются *ИД связывания CPU*.

Изменение системной переменной **_system_configuration.ncpus** влияет на следующее:

- Если после чтения переменной будет удален последний процессор, то **bindprocessor** вернет сообщение об ошибке. Это сообщение впервые появилось при динамическое отключение процессоров (отключении неработающего процессора).
- Приложения, работа которых зависит от числа процессоров, должны считывать значение переменной **_system_configuration.ncpus** после каждого изменения числа процессоров.

Приложения можно также связать с набором процессоров с помощью функции *программных разделов* WLM (WLM). При этом используются ИД логических CPU, которые также начинаются с 0 и заканчиваются на *N*-1. Однако *N* в данном случае - это максимальное число процессоров, поддерживаемых архитектурой раздела. В системе нумерации учитываются как включенные, так и выключенные процессоры.

В силу вышесказанного, при удалении процессоров следует знать, какая именно система нумерации применяется. Число включенных процессоров можно узнать командой **bindprocessor**. Найти процессы и нити, связанные с последним включенным процессором, можно с помощью команды **ps**. Затем с помощью команды **bindprocessor** можно определить новые процессоры.

Для поиска зависимостей WLM необходимо определить конкретный программный раздел, вызвавший ошибку. Для устранения зависимостей выполните следующие действия:

Примечание: Система не связывает с заданиями отключенные или ожидающие отключения процессоры, поэтому если в программном разделе есть другой включенный процессор, то никакие изменения вносить не требуется.

1. Просмотрите список программных разделов, используемых WLM, с помощью команды **lsrset**.
2. Найдите требуемые разделы с помощью команды **lsclass**.
3. Найдите набор классов, использующих эти разделы, с помощью команды **chclass**.
4. Измените классы с помощью команды **wlmctrl**.

После этого в силу вступят новые определения классов, и система автоматически перенесет связанные задания с удаляемого логического процессора.

Понятия, связанные с данным:

“Управление зависимостями DLPAR приложения” на стр. 572
Запрос на удаление DLPAR может вызвать ошибку по нескольким причинам. Наиболее распространенная из

них - это занятость ресурса или нехватка системных ресурсов для выполнения запроса.

Применение операций DLPAR в приложениях

Операции DLPAR могут быть встроены в приложение следующими способами:

- Путем установки набора сценариев DLPAR в каталог. При выполнении операции DLPAR вызываются эти сценарии. Сценарии предназначены для внешней перенастройки приложения.
- С помощью сигнала **SIGRECONFIG**, применяемого для отслеживания сигналов от всех зарегистрированных процессов. При этом предполагается, что приложение умеет отслеживать сигналы, а обработчик сигналов может перенастроить приложение. Обработчик сигнала вызывает специальный интерфейс для определения свойств операции DLPAR.

На высшем уровне в обоих способах применяется одна и та же структура. Для поддержки DLPAR можно использовать любой способ, хотя для управления зависимостями DLPAR, связанными с разделами WLM (наборами процессоров), применяются только сценарии. С WLM не связаны никакие API, поэтому для работы с WLM нельзя использовать обработчики сигналов. Сами по себе, приложения не знают о существовании WLM. Сценарий, запускающий команды WLM для работы с DLPAR, должен вызывать системный администратор.

Выбор конкретного способа должен основываться на архитектуре приложения. Если число нитей или размер буферов приложения настраивается извне, то применим способ со сценариями. Если приложение само отслеживает конфигурацию и ресурсы системы, то рекомендуется использовать сигналы.

Операция DLPAR состоит из нескольких этапов:

- **этап проверки**

  Этап проверки - это первый этап, на котором запрос DLPAR может быть отклонен еще до вмешательства в систему. Например, это может произойти в случае, если при проверке числа лицензий на процессоры будет установлено, что свободных лицензий нет. Кроме того, на этом этапе можно проверить допустимость операции DLPAR для программ, не поддерживающих DLPAR. В некоторых случаях приложение должно указать операторам, что программу необходимо остановить, выполнить запрос, а затем запустить заново.
- **предварительный** и **завершающий**

  этапы*Предварительный* и *завершающий* этапы позволяют остановить программу, выполнить запрос и перезапустить программу.

Перед переходом к следующему этапу DLPAR система пытается убедиться, что для всех ресурсов были полностью выполнены все процедуры проверки кода.

Действия сценариев DLPAR

Сценарии приложения запускаются во время операций добавления и удаления.

При удалении ресурсов сценарии помогают устранить проблемы с приложениями, мешающими удалить ресурс. К числу таких проблем относятся нити, связанные с процессорами, и нехватка закрепленной памяти в системе. Для определения подобных ситуаций существует набор команд, позволяющих составлять сценарии.

Для поиска и устранения зависимостей от DLPAR применяются следующие команды:

- Команда **ps** показывает подключения bindprocessor и состояние системных вызовов **plock** на уровне процесса.
- Команда **bindprocessor** показывает список включенных процессоров и позволяет подключать к ним нити.
- Команда **kill** отправляет процессам сигналы.
- Команда **ipcs** показывает сегменты закрепленной общей памяти на уровне процесса.
- Команда **lsrset** показывает наборы процессоров.
- Команда **lsclass** показывает классы WLM, которые могут содержать наборы процессоров.

- Команда **chclass** позволяет изменить определение класса.

Сценарии также применяются для масштабирования приложений и повышения производительности. При удалении ресурса рекомендуется сократить число нитей и размер буферов приложений. При добавлении ресурсов эти параметры можно увеличить. Данные параметры можно изменять динамически, вызывая команды с помощью сценариев. Сценарии позволяют запустить требуемые команды во время операций DLPAR.

Структура высокого уровня для сценариев DLPAR

В этом разделе приведен обзор сценариев Perl, сценариев оболочки и командных сценариев. Сценарии приложения должны содержать следующие команды:

- **scriptinfo**

  Указывает версию, дату и создается сценария. Может вызываться при установке сценария.
- **register**

  Указывает ресурсы, которыми управляет сценарий. Если сценарий возвращает имя ресурса *cpu*, *mem*, *capacity* или *var_weight*, то он будет автоматически запущен при попытке DLPAR изменить конфигурацию процессоров, памяти, предоставленных ресурсов и переменного веса. Команда register вызывается при установке сценария в подсистеме DLPAR.
- **usage** *имя_ресурса*

  Возвращает информацию об использовании ресурса приложением. На основании этого описания пользователь должен решить, следует ли устанавливать сценарий. Команда должна указывать затрагиваемые функции приложения. Команда **usage** вызывается для всех ресурсов, указанных командой **register** .
- **checkrelease** *имя_ресурса*

  Указывает, должна ли подсистема DLPAR продолжить удаление ресурса. Сценарий не должен удалять ресурс, если важное для системы приложение не поддерживает DLPAR.
- **prerelease** *имя_ресурса*

  Перенастраивает, приостанавливает или прерывает приложение, чтобы оно разблокировало указанный ресурс.
- **postrelease** *имя_ресурса*

  Продолжает или начинает работу приложения.
- **undoprerelease** *имя_ресурса*

  Запускается, если при освобождении ресурса возникла ошибка.
- **checkacquire** *имя_ресурса*

  Указывает, должна ли подсистема DLPAR продолжить добавление ресурса. Она может применяться, например, администратором лицензий, чтобы не допустить добавления нелицензированного ресурса.
- **preacquire** *имя_ресурса*

  Применяется для подготовки к добавлению ресурса.
- **undopreacquire** *имя_ресурса*

  Запускается при ошибке на этапе preacquire и при других сбоях.
- **postacquire** *имя_ресурса*

  Продолжает или начинает работу приложения.
- **preaccevent** *имя-ресурса*

  Применяется для подготовки к обновлению DLPAR.
- **postaccevent** *имя-ресурса*

  Продолжает или начинает работу приложения.
- **undopreaccevent** *имя-ресурса*

  Запускается при ошибке на этапе **preaccevent** и при других сбоях.

- **pretopolgyupdate** *имя-ресурса*

  Применяется для подготовки к обновлению топологии системы.
- **postopolgyupdate** *имя-ресурса*

  Продолжает или начинает работу приложения.

Установка сценариев с помощью команды drmgr

Команда **drmgr** поддерживает внутреннюю базу данных установленных сценариев. Информация для этой базы данных собирается при загрузке системы и обновляется при установке или удалении сценариев. Информацию поставляют команды **scriptinfo**, **register** и **usage**. Для установки сценариев используется команда **drmgr**, она копирует указанные сценарии в хранилище сценариев. По умолчанию, хранилище находится в каталоге **/usr/lib/dr/scripts/all**. В рабочих разделах каталог хранилища по умолчанию — **/var/dr/scripts**. Для хранения информации можно указать и другой каталог. Для определения системы, в которой будет применяться сценарий, укажите при установке сценария имя целевого хоста.

Расположение базового хранилища задается следующей командой:

```
drmgr -R базовый_каталог
```

Для установки сценария введите следующую команду:

```
drmgr -i имя_сценария [-f] [-w минуты] [-D имя_хоста]
```

Определены следующие флаги:

- Флаг **-i** задает имя сценария.
- При замене существующего сценария также применяется флаг **-f**.
- Флаг **-w** указывает ожидаемое время выполнения сценария (в минутах). Этот флаг переопределяет значение, указанное создателем сценария.
- Флаг **-D** позволяет зарегистрировать сценарий для конкретного хоста.

Для удаления сценария введите следующую команду:

```
drmgr -u имя-сценария [-D имя-хоста]
```

Определены следующие флаги:

- Флаг **-u** указывает сценарий, который необходимо удалить.
- Флаг **-D** позволяет удалить сценарий, зарегистрированный в конкретном каталоге.

Для просмотра информации об установленных сценариях введите следующую команду:

```
drmgr -l
```

Соглашение о присвоении имен сценариям

Рекомендуется присваивать сценариям имена, составленные из имени создателя и управляемой подсистемы. Системные администраторы должны присваивать созданным сценариям имена с префиксом *sysadmin*. Например, системный администратор, создавший сценарий для управления WLM, должен назвать его `sysadmin_wlm`.

Среда выполнения и входные параметры сценария

Сценарии запускаются в следующей среде:

- UID процесса равен UID сценария.
- GID процесса равен GID сценария.
- В переменной среды **PATH** указан каталог **/usr/bin:/etc:/usr/sbin**.
- При необходимости может быть задана переменная среды **LANG**.

- Текущий рабочий каталог - **/tmp**.
- Аргументы команды и переменные среды, описывающие событие DLPAR.

Сценарии получают входные параметры в виде аргументов команд и переменных среды, и формируют вывод, записывая пары *имя=значение* в устройство стандартного вывода, причем каждая пара *имя=значение* находится на своей строке. *Имя* указывает имя возвращаемых данных, а *значение* - связанное с именем значение. Текстовые строки могут быть заключены в кавычки, например `DR_ERROR="текст"`. Все переменные среды и пары *имя=значение* должны начинаться со строки `DR_`, зарезервированной для сценариев.

Пара переменных среды **DR_ERROR** *имя=значение* содержит описание ошибки.

Определить этап операции DLPAR, тип действия и тип ресурса, являющегося объектом невыполненного запроса DLPAR, можно с помощью аргументов команды сценария. Например, если указаны аргументы `checkrelease mem`, то этап - `проверка`, действие - `удаление`, а тип ресурса - `память`. Конкретный ресурс можно определить с помощью переменных среды.

При добавлении и удалении памяти используются следующие переменные среды:

Примечание: В следующем описании размер кадра равен 4 Кб.

- **DR_FREE_FRAMES**=`0xFFFFFFFF`

  Число кадров в системе, в шестнадцатеричном формате.
- **DR_MEM_SIZE_COMPLETED**=*n*

  Число успешно добавленных или удаленных мегабайт, в десятичном формате.
- **DR_MEM_SIZE_REQUEST**=*n*

  Объем памяти, указанный в запросе; в десятичном формате.
- **DR_PINNABLE_FRAMES**=`0xFFFFFFFF`

  Общее число закрепляемых кадров в системе, в шестнадцатеричном формате. Этот параметр применяется при удалении памяти и позволяет вычислить объем свободной закрепляемой памяти, нехватка которой вызывает большую часть ошибок.
- **DR_TOTAL_FRAMES**=`0xFFFFFFFF`

  Общее число кадров памяти в системе, в шестнадцатеричном формате.

При добавлении и удалении процессора используются следующие переменные среды:

- **DR_BCPUID**=*N*

  ИД связывания удаляемого или добавляемого процессора, в десятичном формате. Подключение к этому процессору с помощью **bindprocessor** не обязательно означает, что подключение требуется прервать. Это справедливо только для процессора с номером *N*, поскольку именно он всегда удаляется при удалении процессора. ИД связывания непрерывны, лежат в диапазоне от 0 до *N* и предназначены для идентификации включенных процессоров. Число включенных процессоров можно узнать с помощью команды **bindprocessor**.
- **DR_LCPUID**=*N*

  Логический ИД удаляемого или добавляемого процессора, в десятичном формате.

При работе с Micro-Partitioning применяются следующие переменные среды.

DR_CPU_CAPACITY=N
: Процент общих физических процессоров в разделе.

DR_VAR_WEIGHT=N
: Относительный приоритет раздела при распределении циклов простоя общего пула.

DR_CPU_CAPACITY_DELTA=N
Разность между текущим процентом общих физических процессоров, предоставленных разделу, и процентом общих физических процессоров, которые будут предоставлены разделу после завершения данной операции.

DR_VAR_WEIGHT_DELTA=N
Разность между текущим процентным переменным весом раздела и процентным переменным весом, который будет назначен разделу после завершения данной операции.

Оператор может просмотреть сведения о текущем запросе DLPAR с помощью HMC с заданным уровнем подробности. Этот параметр передается сценарию с помощью переменной среды **DR_DETAIL_LEVEL**=*N*, где *N* - число от `0` до `5`. Значение по умолчанию 0 указывает, что информация предоставлена не будет. Значение 1 зарезервировано операционной системой и используется для сообщений высокого уровня. Оставшиеся уровни (2-5) могут применяться сценариями, причем большие уровни обеспечивают большую подробность.

Сценарии передают данные, записывая в стандартный вывод следующие пары *имя=значение*:

Пара имя=значение	Описание
DR_LOG_ERR=*сообщение*	Отправляет сообщение с уровнем syslog, равным значению переменной среды **LOG_ERR**.
DR_LOG_WARNING=*сообщение*	Отправляет сообщение с уровнем syslog, равным значению переменной среды **LOG_WARNING**.
DR_LOG_INFO=*сообщение*	Отправляет сообщение с уровнем syslog, равным значению переменной среды **LOG_INFO**.
DR_LOG_EMERG=*сообщение*	Отправляет сообщение с уровнем syslog, равным значению переменной среды **LOG_EMERG**.
DR_LOG_DEBUG=*сообщение*	Отправляет сообщение с уровнем syslog, равным значению переменной среды **LOG_DEBUG**.

Оператор также может настроить занесение информации в протокол с помощью средства **syslog**. В этом случае все описанные выше сообщения будут также направлены этому средству. Для работы с этой функцией вам потребуется настроить средство **syslog**.

DLPAR команды сценариев

В этом разделе описаны следующие команды сценариев для DLPAR:

`scriptinfo`
Предоставляет информацию об установленных сценариях, такую как дата создания и ресурсы.

`register`
Показывает список ресурсов, которыми управляет сценарий. Затем с помощью команды **drmgr** можно запустить сценарии для конкретного типа ресурса.

`usage`
Предоставляет информацию об управлении ресурсом в удобном для чтения формате. На основании этой информации пользователь должен определить, какое влияние окажет изменение ресурса на приложения и службы. Эта команда запускается при установке сценария; предоставляемая ей информация хранится во внутренней базе данных для команды **drmgr**. Для ее просмотра воспользуйтесь опцией *-l* команды **drmgr**.

`checkrelease`
Команда **drmgr** позволяет оценить последствия удаления ресурсов на систему. Обычно для этого вызываются сценарии DLPAR, содержащие команду **checkrelease**. Каждый сценарий DLPAR может оценить особенности приложения и указать команде **drmgr**, использующей код возврата сценария, повлияет ли удаление ресурса на приложения. Если удаление допустимо, то будет возвращен код успешного выполнения. Если текущее состояние приложения не позволяет удалить ресурс, то сценарий

вернет сообщение об ошибке. Если пользователем указана опция *FORCE*, применимая ко всем этапам операции DLPAR, то команда **drmgr** пропустит этап **checkrelease** и сразу перейдет к командам **prerelease**.

prerelease
Перед удалением ресурса сценарии DLPAR должны полностью исключить или снизить нагрузку на него. Если этого сделать не удается, то должно быть возвращено сообщение об ошибке. В любом режиме выполнения система попытается удалить ресурс и вызовет сценарий завершающего этапа, вне зависимости от действий, предпринятых или не предпринятых командой **prerelease**. Предпринимаемые операционной системой действия безопасны. Если ресурс нельзя безопасно удалить, то операция выполнена не будет.

Сценарий DLPAR должен сохранять информацию о действиях, предпринятых командой **prerelease**, чтобы восстановить их в случае ошибки на завершающем этапе. Если применяется повторное обнаружение, то эти действия можно предпринять на завершающем этапе. Опцию *force* следует использовать с особой осторожностью.

postrelease
После освобождения ресурса для каждого установленного сценария DLPAR запускается команда **postrelease**. На этом этапе сценарии DLPAR выполняют все завершающие действия. Прерванные приложения должны быть перезапущены.

Вызывающая программа игнорирует все ошибки, о которых сообщают команды **postrelease**, и операция будет считаться успешной даже в том случае, если пользователь был уведомлен об ошибках. Для сообщения о том, что приложение не было перенастроено, служит переменная среды **DR_ERROR**.

undoprerelease
Если в результате вызова команды **prerelease** командой **drmgr** из сценария DLPAR возникли ошибки, то команда **drmgr** попытается восстановить исходное состояние. Для этого команда **drmgr** вызовет в сценарии DLPAR команду **undoprerelease**. Команда **undoprerelease** запускается только в том случае, если ранее сценарий пытался освободить ресурсы для запроса DLPAR. На данном этапе сценарий должен отменить все изменения, внесенные командой **prerelease**. Для этого сценарий должен сохранять информацию обо всех выполняемых действиях и состоянии системы или иным образом обеспечить возможность восстановления сведений о состоянии системы и повторной настройки приложения, как если бы, фактически, событие DLPAR никогда не имело места.

checkacquire
Это первая команда сценария DLPAR, вызываемая при получении нового ресурса. Она вызывается всеми сценариями, поддерживающими добавление данного типа ресурсов. Одним из применений этапа checkacquire является проверка лицензий на процессоры администраторами лицензий. Команда **checkacquire** запускается всегда, вне зависимости от значения переменной среды **FORCE**, и вызывающая программа должна учитывать ее код возврата. Пользователь не может принудительно добавить процессор, если сценарием DLPAR или программой, поддерживающей DLPAR, не пройден этап проверки.

Таким образом, переменная среды **FORCE** неприменима к команде **checkacquire**, хотя она и используется на остальных этапах. На этапе preacquire она указывает, насколько серьезно сценарий может вмешиваться в конфигурацию приложения. Опция *force* может применяться сценариями для управления стратегией остановки и перезапуска приложений, аналогично освобождению ресурсов с поддержкой DLPAR.

preacquire
Если на этапе checkacquire не возникло ошибок, система переходит к этапу preacquire, на котором выполняется набор сценариев, подготавливающих добавление ресурса с помощью команды **preacquire**. Если пользователем не задана переменная среды **FORCE** и не возникло никаких ошибок, то выполняются все указанные сценарии. Если задана переменная среды **FORCE**, то система переходит к этапу добавления вне зависимости от кода возврата. Если задана переменная среды **FORCE**, то ошибки не учитываются, поскольку их можно избежать путем перенастройки приложения, что допустимо при применении переменной среды **FORCE**. Если возникла ошибка, а переменная **FORCE** не указана, то

система переходит к этапу undopreacquire и отменяет все выполненные к этому моменту сценарии. На этом этапе сценарии должны отменить все внесенные изменения.

undopreacquire
Этап undopreacquire позволяет отменить все внесенные изменения. Если сценарий переходит к этапу undopreacquire, то предполагается, что он успешно выполнил команду **preacquire**.

postacquire
Команда **postacquire** вызывается после успешного добавления ресурса в систему. При этом снова вызываются все сценарии DLPAR, вызывавшиеся ранее на этапах проверки и подготовки. Эта команда позволяет встроить новый ресурс в приложение. Например, приложение может создать новые нити или увеличить буферы. Прерванные ранее приложения должны быть перезапущены.

checkmigrate
Это первая команда сценария DLPAR, вызываемая при переносе. Она вызывается всеми сценариями, поддерживающими добавление данного типа ресурсов. Команда **checkmigrate** запускается всегда, вне зависимости от значения переменной среды **FORCE**, и вызывающая программа должна учитывать ее код возврата. Пользователь не может принудительно перенести раздел, если сценарием или программой DLPAR-aware не пройден этап проверки.

premigrate
Если на этапе **checkmigrate** не возникло ошибок, система переходит к этапу **premigrate**, на котором запускается набор сценариев, подготавливающих раздел. Все указанные сценарии выполняются до переноса раздела. Система переходит к состоянию переноса независимо от кода возврата сценария. Если возникла ошибка, то система переходит к этапу **undopremigrate** и отменяет все выполненные к этому моменту сценарии. На этом этапе сценарии должны отменить все внесенные изменения.

undopremigrate
Этап **undopremigrate** позволяет отменить все внесенные изменения. Если сценарий переходит к этапу **undopremigrate**, то предполагается, что он успешно выполнил команду **premigrate**.

postmigrate
Команда **postmigrate** вызывается после успешного переноса раздела. При этом снова вызываются все сценарии DLPAR, вызывавшиеся ранее на этапах проверки и подготовки.

pretopologyupdate
Команда **pretopologyupdate** выполняется до выполнения действий, затрагивающих изменение топологии (например, добавление или удаления процессоров и памяти). Данная команда предназначена для сообщения сценариям о начале операции изменения топологии, которая не должна завершиться неудачно. Система переходит к этапу добавления вне зависимости от кода возврата.

posttopologyupdate
Команда **posttopologyupdate** вызывается после успешного завершения операции изменения топологии. При этом снова вызываются все сценарии DLPAR, вызывавшиеся ранее на этапах подготовки.

checkhibernate
Это первая команда сценария DLPAR, вызываемая при отключении с сохранением состояния. Она вызывается всеми сценариями, поддерживающими добавление данного типа ресурсов. Команда **checkhibernate** запускается всегда, вне зависимости от значения переменной среды **FORCE**, и вызывающая программа должна учитывать ее код возврата. Пользователь не может принудительно отключить раздел с сохранением состояния, если сценарием или программой DLPAR-aware не пройден этап проверки.

prehibernate
Если на этапе **checkhibernate** не возникло ошибок, система переходит к этапу **prehibernate**, на котором запускается набор сценариев, подготавливающих раздел. Все указанные сценарии выполняются до отключения с сохранением состояния. Система переходит к состоянию отключения с сохранением состояния независимо от кода возврата сценария. Если возникла ошибка, то система переходит к этапу **undohibernate** и отменяет все выполненные к этому моменту сценарии. На этом этапе сценарии должны отменить все внесенные изменения.

undohibernate
Этап **undohibernate** позволяет отменить все внесенные изменения. Если сценарий переходит к этапу **checkhibernate**, то предполагается, что он успешно выполнил команду **checkhibernate**.

posthibernate
Команда **posthibernate** вызывается после успешного отключения раздела с сохранением состояния. При этом снова вызываются все сценарии DLPAR, вызывавшиеся ранее на этапах проверки и подготовки.

preaccevent
Это первая команда сценария DLPAR, вызываемая в последовательности ускорителя шифрования DLPAR. Она вызывается всеми сценариями, поддерживающими добавление или освобождение данного типа ресурсов. В момент ее выполнения неизвестен тип следующего действие: добавление или освобождение ускорителя шифрования. Действие указывается на одном из следующих этапов.

postaccevent
Команда **postaccevent** вызывается после успешного добавления ресурса в систему. При этом снова вызывается каждый сценарий DLPAR, который вызывался ранее на этапе подготовки. Эта команда позволяет встроить новое состояние ресурса в приложение.

undoaccevent
Этап **undoaccevent** позволяет отменить все внесенные изменения. Если сценарий переходит к этапу **undoaccevent**, то он успешно выполнил команду **preaccevent**.

Настройка расширений ядра для поддержки DLPAR

Как и приложения, расширения ядра по умолчанию допускают DLPAR.

Однако некоторые из них могут зависеть от конфигурации системы и должны быть зарегистрированы в подсистеме DLPAR. Некоторые расширения ядра размещаются данные или создают нити на основании числа процессоров, либо создают большие пулы буферов с закрепленной памятью. Такие расширения необходимо уведомить об изменении конфигурации системы. Механизм и действия, которые необходимо предпринять, аналогичны действиям, предпринимаемым для приложений, поддерживающих DLPAR.

Регистрация обработчиков перенастройки

Для регистрации и отмены регистрации обработчиков перенастройки предусмотрены следующие службы:

```
#include sys/dr.h

int reconfig_register(int (*handler)(void *, void *, int, dr_info_t *),
                      int actions, void * h_arg, ulong *h_token, char *name);

void reconfig_unregister(ulong h_token);

int (*handler)(void *event, void *h_arg, unsigned long long req, void *resource_info);

void reconfig_unregister(ulong h_token);

int reconfig_register_ext (int (*handler)(void *, void *, unsigned long long, dr_info_t *),
unsigned long long actions, void * h_arg, ulong *h_token, char *name);

int (*handler)(void *event, void *h_arg, unsigned long long req, void *resource_info);

kerrno_t reconfig_register_list(int (*handler)(void *, void *, dr_kevent_t, void *),
dr_kevent_t event_list[], size_t list_size, void *h_arg, ulong *h_token, char *name);

int (*handler)(void *event, void *h_arg, dr_kevent_t event_in_prog, void *resource_info);
```

Примечание: Можно использовать службу ядра **reconfig_register_list**. Эта служба поддерживает извещение расширений ядра о большем числе событий. Прежние службы ядра **reconfig_register** и **reconfig_register_ext**) ограничены 32 и 64 событиями, соответственно. Это не позволяет перенести расширения ядра, использующие данную службу, в более поздние системы, поддерживающие больше 32 и 64 событий.

Функции **reconfig_register**, **reconfig_register_ext** и **reconfig_register_list** имеют следующие параметры:

- Параметр *handler* - это вызываемая функция расширения ядра.
- Параметр *actions* позволяет указать, для каких событий требуется уведомление. Список событий приведен в описании служб ядра **reconfig_register**, **reconfig_register_ext** и **reconfig_unregister**.
- Параметр *h_arg* указывается расширением ядра, хранится вместе с дескриптором функции и передается обработчику при его вызове. Он не используется ядром напрямую, но позволяет поддерживать расширения ядра, управляющие несколькими экземплярами адаптера. Фактически этот параметр указывает на блок управления адаптером.
- Выходной параметр *h_token* используется при отмене регистрации обработчика.
- Информационный параметр *name* может быть включен в сообщение об ошибке драйвера. Он указывается расширением ядра и должен содержать не более 15 символов ASCII.
- Параметр *event_list* является массивом значений dr_kevent_t, для событий, уведомления о которых должны быть направлены расширению ядра. Список определенных событий приведен в описании службы ядра **reconfig_register_list**.
- Параметр *list_size* - это объем памяти, необходимый для параметра *event_list*.

Функции **reconfig_register** и **reconfig_register_ext** возвращают `0` при успешном завершении и соответствующее значение `errno` - в случае ошибки.

Функция **reconfig_unregister** удаляет установленный ранее обработчик.

Функции **reconfig_register**, **reconfig_register_ext** и **reconfig_unregister** можно вызывать только в среде процесса.

Если расширение ядра регистрируется на предварительном этапе, то рекомендуется также зарегистрировать его на этапе проверки, чтобы избежать ошибок в конфигурации при удалении ресурсов.

Обработчики перенастройки

Обработчик перенастройки, используемый со службой ядра **reconfig_register_list**, имеет следующий интерфейс:

```
Int (*handler)(void *event, void *h_arg, dr_kevent_t event_in_prog, void *resource_info);
```

Обработчик перенастройки имеет следующие параметры:

- Параметр *event* передается обработчику для использования при вызове функции **reconfig_handler_complete**.
- Параметр *h_arg* указывается обработчиком при регистрации.
- Параметр *event_in_prog* указывает операцию DLPAR, выполняемую обработчиком. Список событий приведен в описании службы ядра **reconfig_register_list**.
- Параметр *resource_info* задает информацию о ресурсе для текущего запроса DLPAR. Если запрос основывается на процессе, то данные *resource_info* передаются через структуру *dri_cpu*. Если процесс основывается на памяти, применяется структура *dri_mem*. В разделе Micro-Partitioning, если запрос основан на емкости процессора, данные *resource_info* предоставляются посредством структуры *dri_cpu_capacity*. Дополнительная информация о структуре *dri_cpu_capacity* и о ее формате приведена в разделе Служба ядра reconfig.

```
struct dri_cpu {
        cpu_t           lcpu;           /* ИД логического или целевого CPU */
        cpu_t           bcpu;           /* ИД связывания целевого CPU       */
};

struct dri_mem {
        size64_t        req_memsz_change;   /* запрошенный пользователем объем памяти */
        size64_t        sys_memsz;          /* начальный объем памяти системы */
        size64_t        act_memsz_change;   /* удаленная или добавленная память */
        rpn64_t         sys_free_frames;    /* число свободных кадров */
```

```
        rpn64_t         sys_pinnable_frames;/* число закрепляемых кадров */
        rpn64_t         sys_total_frames;   /* общее число кадров */
        unsigned long long lmb_addr;        /* начальный адрес логического блока памяти */
        size64_t        lmb_size;           /* размер добавляемого логического блока памяти */
};
```

Если текущий запрос DLPAR является переносом раздела, обработчик предоставляет данные *resource_info* расширениям ядра *resource_info*, однако расширениям ядра не требуется доступ к содержимому данных *resource_info*, поскольку эти данные расширениями ядра не используются.

Обработчики перенастройки вызываются в среде процесса.

Для расширений ядра применимы следующие ограничения:

- В каждый момент времени может быть перенастроен только один типа ресурса.
- Одновременно нельзя указывать несколько процессоров. Расширения ядра должны уметь обрабатывать добавление и удаление нескольких логических блоков памяти. Можно отправить запрос на добавление или удаление нескольких гигабайт памяти.

Этап проверки позволяет приложениям, поддерживающим DLPAR, проанализировать запрос пользователя перед его выполнением. Соответствующее расширение ядра вызывается только один раз, даже если получен запрос на удаление нескольких блоков памяти. В то же время, предварительный и завершающий этапы, а также этап обработки ошибки выполняются для каждого блока памяти. Обратите внимание, что при уведомлении приложений эта три этапа выполнялись только один раз для каждого запроса. Другое отличие заключается в том, что этап обработки ошибки для расширения ядра выполняется при сбое любого блока памяти, а для приложения - при сбое всего пользовательского запроса.

В общем случае, на этапе проверки расширение ядра проверяет состояние системы и убеждается, что оно допустимо для выполнения поступившего запроса DLPAR. При отрицательном результате обработчик возвращает **DR_FAIL**. В противном случае, возвращается **DR_SUCCESS**.

На предварительном этапе операции удаления расширения ядра пытаются удалить все оставшиеся зависимости от указанного ресурса. Примером может служить драйвер, управляющий пулами буферов для каждого процессора. Драйвер может пометить связанные пулы как ожидающие удаления, чтобы исключить выделение в них памяти для новых запросов. Через некоторое время пулы можно будет очистить. Другими объектами, учитываемыми на предварительном этапе удаления, являются таймеры и связанные нити, которые должны быть, соответственно, остановлены и прерваны. Кроме того, может быть удалена связь с нитями.

На завершающем этапе операции удаления расширения ядра пытаются очистить ресурсы с помощью операции сбора мусора, предполагая, что ресурс действительно удален. В противном случае, таймеры и нити должны быть установлены заново. Запрос **DR_resource_POST_ERROR** применяется для сообщения об ошибке.

На предварительном этапе добавления расширения ядра должны инициализировать все пути к данным, чтобы настроенный ресурс можно было использовать. Система не гарантирует, что к ресурсу не поступит обращений до того, как обработчик будет снова вызван на завершающем этапе.

На завершающем этапе удаления расширения ядра должны убедиться, что ресурс был правило добавлен, и теперь может использоваться. На этом этапе рекомендуется запускать связанные нити, настраивать таймеры и увеличивать размер буферов.

Расширения ядра можно также уведомить об удалениях или добавлениях памяти при каждой операции (в значительной степени, подобно приложениям), зарегистрировав один или несколько типов извещений **_OP_**. Это позволяет расширению ядра модифицировать использование ресурсов в ответ на операцию DR с памятью один раз для каждой операции, а не для каждого логического блока памяти (LMB).

Уведомление **DR_MEM_REMOVE_OP_PRE** отправляется перед удалением памяти. Обработчики перенастройки могут начать регулировку своих ресурсов в ожидании удаления памяти. Уведомления **DR_MEM_REMOVE_OP_POST** и **DR_MEM_ADD_OP_POST** отправляются после операций удаления или добавления памяти соответственно, независимо от того, была ли операция выполнена успешно. В случае сбоя операции **act_memsz_change** равно `0`.

Обычно в течение нескольких секунд обработчики перенастройки возвращают значение **DR_SUCCESS** при успешном выполнении и **DR_FAIL** при сбое. Если требуется больше времени, то возвращается **DR_WAIT**.

Расширенные обработчики DR

Если расширение ядра не может выполнить операцию за несколько секунд, то оно возвращает значение **DR_WAIT** и продолжает обработку запроса в асинхронном режиме. При завершении запроса вызывается функция **reconfig_handler_complete**.

```
void reconfig_handler_complete(void *event, int rc);
```

Параметр *event* был передан обработчику при вызове. Параметр *rc* должен быть равен либо **DR_SUCCESS** при успешном завершении, либо **DR_FAIL** при сбое.

Службу ядра **reconfig_handler_complete** можно вызывать в среде процесса или прерывания.

Применение службы ядра xmemdma

В системах, поддерживающих операции DLPAR, например, динамическое удаление памяти, вызов службы ядра **xmemdma** без флага **XMEM_DR_SAFE** приводит к тому, что указанная область памяти помечается как недоступная для удаления. Это применяется для обеспечения целостности системы, так как у системы нет информации о том, каким образом инициатор планирует использовать возвращенный адрес физической памяти. Операции динамического удаления памяти можно применять по отношению ко всей памяти, за исключением той, которая была указана в вызове **xmemdma**.

Если инициатор планирует использовать адрес физической памяти только в информационных целях (например, для буферов трассировки или отладочной информации), он может задать флаг **XMEM_DR_SAFE**. Этот флаг сигнализирует системе о том, что адрес физической памяти можно предоставить инициатору безо всякого риска повреждения данных. При наличии такого флага система разрешает динамически удалять указанную память.

Если инициатор планирует применять адрес физической памяти для фактического доступа к данным путем выключения преобразования данных и использования функций доступа к физической памяти CPU вида загрузить/сохранить, либо путем создания контроллеров прямого доступа к памяти (DMA) для работы с физической памятью, то флаг **XMEM_DR_SAFE** указывать не нужно. При наличии такого флага операция динамического удаления памяти может привести к нарушению целостности данных системы. Информацию о преобразовании расширений ядра, использующих физическую память описанным способом, в версию, поддерживающую DLPAR, можно получить в сервисном представительстве IBM.

Более подробные сведения по этому вопросу приведены в описании службы ядра **xmemdma**.

Управление уведомлением приложений об операциях над памятью DLPAR

Динамическое добавление и удаление памяти логических разделов, в которых работают программы с поддержкой DLPAR, может привести к конфликтам ресурсов. По умолчанию каждая программа получает уведомления об изменениях ресурсов. Например, удаление 1 Гб памяти из раздела, в котором выполняются две программы, поддерживающие DR, приведет к отправке каждой из этих программ соответствующего уведомления. Поскольку программы работают независимо друг от друга, то каждая из них уменьшит объем доступной памяти на 1 Гб, что приведет к снижению эффективности. Аналогичная неполадка может возникнуть при добавлении памяти.

Для того чтобы избежать этой неполадки AIX разрешает установку сценариев, вычисляющих коэффициент, указывающий на процентную долю фактического изменения памяти. Система уведомляет приложение в случае выполнения операции DLPAR по изменению памяти. Данный коэффициент можно указать с помощью пары **DR_MEM_PERCENT имя=значение** в ходе установки сценария с помощью команды **drmgr**. Эта пара имя=значение применяется при вызове сценария с помощью команды **drmgr scriptinfo**. Допустимы целые значения в диапазоне от 1 до 100. Остальные значения игнорируются и вместо них указывается значение по умолчанию 100. Кроме того, данную пару **имя=значение** можно указать в качестве переменной среды в процессе установки. В этом случае значение переменной среды, если оно задано, переопределяет значение, указанное сценарием.

Аналогичным образом при работе с приложениями, использующими обработчик сигналов **SIGRECONFIG** и системный вызов **dr_reconfig()**, вы можете управлять уведомлениями об операциях DLPAR по изменению памяти, указав пару **DR_MEM_PERCENT имя=значение** в качестве переменной среды. Обратите внимание, что для изменения этого значения требуется перезапуск приложения.

Программа sed

Программа **sed** - это текстовый редактор, функции которого схожи с функциями строчного редактора **ed**.

Однако, в отличие от **ed**, программа **sed** редактирует текст автономно, а не в интерактивном диалоге с пользователем.

Обработка строк с помощью sed

Программа **sed** работает без вмешательства пользователя, запросившего редактирование.

В таком режиме работы **sed** может выполнять следующие действия:

- Редактировать большие файлы.
- Выполнять сложные операции редактирования несколько раз, не запрашивая (в отличие от интерактивных редакторов) дополнительного ввода информации и позиционирования курсора.
- Выполнять глобальные изменения за один проход.

Редактор хранит в памяти лишь небольшое количество строк редактируемого файла и не создает временных файлов. Поэтому длина редактируемого файла ограничена только объемом памяти, доступной для входного файла и вывода.

Понятия, связанные с данным:

“Инструменты и утилиты” на стр. 1
В этом разделе приведен обзор инструментов и утилит, предназначенных для разработки программ на языке C.

Запуск редактора

Ниже приведен файл описания для обновления программы **make**.

Каждая команда в командном файле должна быть указана на отдельной строке. После создания этого файла введите в командной строке:

```
sed -fкомандный-файл >вывод <ввод
```

У этой команды есть следующие параметры:

Параметр	Определение
командный_файл	Имя командного файла, содержащего команды редактирования.
Вывод	Имя файла, в который будет помещен отредактированный вывод.
Ввод	Имя файла (или файлов) для редактирования.

После этого программа **sed** выполняет необходимые изменения и записывает полученную информацию в файл вывода. Содержимое входного файла остается без изменений.

Алгоритм работы программы sed

Программа **sed** - это потоковый редактор, получающий данные из стандартного ввода, изменяющий их в соответствии с инструкциями из командного файла и записывающий результирующий поток в стандартный вывод.

Если вы не создадите командный файл и не укажете флаги для команды **sed**, то программа **sed** скопирует стандартный ввод в стандартный вывод без изменений. Входные данный для этой программы поступают из двух источников:

Программа	Описание
Поток ввода	Поток символов ASCII из файлов или непосредственно с клавиатуры. Эти данные будут отредактированы программой.
Команды	Набор адресов и связанных команд, указанных для выполнения в следующем формате: `[Строка1 [,Строка2] ] команда [аргумент]` Параметры *Строка1* и *Строка2* называются адресами. Они могут представлять собой шаблоны, по которым будет выполнен поиск во входных данных, или номера строк входного потока.

Команды редактирования можно вводить вместе с командой **sed** с помощью флага **-e**.

Команда **sed** построчно считывает входной поток данных в область памяти, называемую областью шаблона. После помещения строки данных в область шаблона **sed** считывает командный файл и сравнивает адреса в командном файле с символами этой строки. Если будет встречено совпадение, то **sed** выполнит связанную с данным адресом команду над символами области шаблона. В результате выполнения команды содержимое области шаблона будет изменено и передано на вход последующих команд.

Сравнив все адреса, указанные в командном файле, с содержимым области шаблона, программа **sed** записывает итоговое содержимое области шаблона в стандартный вывод. После этого из стандартного ввода считывается следующая строка и повторяется процесс, задаваемый командным файлом.

Некоторые команды редактирования изменяют описанную последовательность действий.

На способ выполнения команды **sed** влияют и заданные вместе с ней флаги.

Работа с регулярными выражениями

Регулярное выражение - это строка, содержащая обычные символы, символы подстановки и/или операторы, задающие набор вариантов строк.

В редакторе потока используются те же символы подстановки, что и в редакторе **ed**; они отличаются от символов подстановки оболочки.

Обзор команд sed

У всех команд **sed** однобуквенные имена. Обычно они вызываются с некоторыми параметрами, например, номерами строк или текстовыми строками.

Перечисленные ниже команды изменяют строки в области шаблона.

В синтаксических диаграммах применяются следующие символы:

Символ	Значение
[ ]	В квадратные скобки заключаются необязательные параметры команды.
курсив	Параметры, выделенные курсивом, обозначают значения, которые вы должны ввести. Например, параметр *имя_файла* нужно заменить фактическим именем файла.
Строка1	Этот параметр обозначает номер строки ввода или стандартное выражение для сравнения, с которого начнется применение команды редактирования.
Строка2	Этот параметр обозначает номер строки ввода или стандартное выражение для сравнения, на котором закончится применение команды редактирования.

Обработка строк

В этом разделе рассмотрена обработка строк.

Функция	Синтаксис/Описание
добавить строки	[*Строка1*]**a****n***Текст* Записывает строки, содержащиеся в блоке *Текст*, в поток вывода после *Строки1*. Команда **a** указывается в конце строки.
изменить строки	[*Строка1* [,*Строка2*]]**c****n***Текст* Удаляет строки с адресами *Строка1* и *Строка2*, как и команда *удалить строки*. Затем записывает вместо них в поток вывода блок *Текст*.
удалить строки	[*Строка1* [,*Строка2*]]**d** Удаляет строки из потока ввода без последующего копирования их в поток вывода. Будут удалены строки, начиная со *Строки1*. Следующей в поток вывода будет скопирована строка с номером *Строка2* + 1. Если вы укажете только один номер строки, то будет удалена только одна указанная строка. Если вы не укажете номер строки, то следующая строка не будет скопирована. Над строками, не скопированными в поток вывода, никакие действия выполнить нельзя.
вставить строки	[*Строка1*] i **n***Текст* Записывает строки из блока *Текст* в поток вывода перед *Строкой1*. Команда **i** указывается в конце строки.
следующая строка	[*Строка1* [,*Строка2*]]**n** Считывает следующую строку или группу строк, начиная от *Строки1* и заканчивая *Строкой2*, в область шаблона. Текущее содержимое области шаблона будет записано в вывод (если оно не было удалено).

Подстановка

В этом разделе рассмотрена подстановка.

Функция	Синтаксис/Описание
подстановка по шаблону	[*Строка1* [,*Строка2*]] **s**/*Шаблон*/*Строка*/*Флаги* Выполняет в заданной строке (строках) поиск последовательности символов, соответствующей регулярному выражению, заданному в параметре *Шаблон*. Если такие символы будут найдены, команда заменит их другим набором символов, задаваемых параметром *Строка*.

Ввод и вывод

В этом разделе рассмотрены функции ввода-вывода.

Функция	Синтаксис/Описание
печать строк	[*Строка1* [,*Строка2*]] **p** Записывает указанные строки в STDOUT с теми изменениями, которые были внесены в ходе редактирования до вызова команды **p**.
записать строки	[*строка-1* [,*строка-2*]]**w***имя-файла* Записывает в файл *имя-файла* указанные строки с теми изменениями, которые были внесены в ходе редактирования до вызова команды **w**. Если файл с именем *имя-файла* существует, то он обновляется; в противном случае он создается. В процессе редактирования можно задать до 10 различных файлов ввода или вывода. Между именем команды **w** и параметром *имя_файла* должен быть указан строго один пробел.
считать файл	[*строка-1*]**r***имя-файла* Считывает содержимое файла *имя-файла* и добавляет его после *строки-1*. Между именем команды **r** и *именем-файла* должен быть указан ровно один пробел. Если файл *имя-файла* открыть не удастся, то команда будет считать, что файл пустой. Сообщение об ошибке выдано не будет.

Поиск шаблона с объединением строк

В этом разделе рассмотрено сравнение строк.

Функция	Синтаксис/Описание
объединить строки	[*Строка1* [,*Строка2*]]**N** Объединяет указанные строки ввода, подставляя вместо разрыва символ новой строки. Шаблон может содержать символы из обоих строк.
удалить первую строку области шаблона	[*Строка1* [,*Строка2*]]**D** Удаляет весь текст в области шаблона до первого символа новой строки (включительно). Если в области шаблона записана только одна строка, то будет считана следующая строка. Повторно выполняет все указанные команды редактирования с самого начала.
печатать первую строку области шаблона	[*Строка1* [,*Строка2*]]**P** Записывает весь текст из области шаблона до первого символа новой строки (включительно) в STDOUT.

Копирование и вставка

В этом разделе рассмотрены операции копирования и вставки.

Функция	Синтаксис/Описание
скопировать	[*Строка1* [,*Строка2*]]**h** Копирует текст из области шаблона, расположенный между *Строкой1* и *Строкой2*, в промежуточную область.
скопировать с добавлением	[*Строка1* [,*Строка2*]]**H** Копирует текст из области шаблона, расположенный между *Строкой1* и *Строкой2*, в промежуточную область путем добавления к хранящейся там информации.
вставить копию	[*Строка1* [,*Строка2*]]**g** Копирует текст из промежуточной области в область шаблона, вставляя его между *Строкой1* и *Строкой2*. Содержимое, уже хранящееся в области шаблона, будет уничтожено.

Функция	Синтаксис/Описание
вставить копию с добавлением	[*Строка1* [,*Строка2*]]**G** Копирует текст из промежуточной области в область шаблона после блока, ограниченного *Строкой1* и *Строкой2*. Содержимое, уже хранящееся в области шаблона, изменено не будет. От добавленного текста его будет отделять символ новой строки.
обмен копиями	[*Строка1* [,*Строка2*]]**x** Выполняет обмен содержимого промежуточной области и текста из области шаблона, расположенного между *Строкой1* и *Строкой2*.

Управление

В этом разделе рассмотрены операции копирования и вставки.

Функция	Синтаксис/Описание
отрицание	[*Строка1* [,*Строка2*]]**!** Символ **!** (восклицательный знак) означает, что указанная после него команда должна быть применена к данным файла ввода, лежащим *вне* области, ограниченной строками *Строка1* и *Строка2*.

группы команд	[*Строка1* [,*Строка2*]]{
	
	набор команд
	
	}
	
	В фигурных скобках ({ }) указывается набор команд, которые должны быть выполнены над строками ввода, расположенными между *Строкой1* и *Строкой2*. Первая команда из этого набора должна быть указана в одной строке с левой скобкой или в следующей после нее строке. Правая скобка должна указываться в отдельной строке. Допускается вложенность групп команд.

метки	:*Метка* Отмечает для каждой ветви выполнения конечную точку в последовательности команд редактирования. *Метка* представляет собой строку длиной до 8 символов. Все *Метки* в потоке редактирования должны быть уникальными.
безусловный переход к метке	[*Строка1* [,*Строка2*]]**b***Метка* Переход к команде редактирования с заданной *меткой*. Обработка текущего ввода продолжится со следующей команды. Если задана пустая *Метка*, то будет выполнен переход к концу последовательности команд редактирования, после чего будет считана новая строка ввода, и для нее эта последовательность начнет выполняться сначала. Указываемая здесь *метка* должна быть предварительно задана в последовательности команд.
переход с проверкой	[*Строка1* [,*Строка2*]]**t***Метка* Если в текущей строке ввода была выполнена хотя бы одна подстановка, то переход к команде с заданной *Меткой*. Если подстановка не выполнялась, то команда никаких действий не выполнит. Флаг, установленный после выполнения подстановки, сбрасывается. Этот флаг сбрасывается в начале каждой новой строки ввода.
wait	[*Строка1*]**q** Останавливает процесс редактирования следующим образом: записывает в поток вывода текущую строку и данные о проверке добавления или чтения, после чего редактор останавливается.

Функция	Синтаксис/Описание
определить номер строки	[*Строка1*]**=** Записывает в стандартный вывод номер строки, совпадающей со *Строкой1*.

Применение текста в командах

Для команд обработки строк **добавить**, **вставить** и **изменить** задается текст, который будет добавлен в поток вывода.

Этот текст задается в соответствии со следующими правилами:

- Его длина может составлять как одну, так и несколько строк.
- Каждый символ новой строки (\n), включенный в *Текст*, должен предваряться дополнительным символом \ (\\n).
- Строка *Текст* завершается обычным символом новой строки, без дополнительной косой черты \ (\n).
- После вставки строки *Текст* она:
  - Всегда записывается в поток вывода, независимо от действий других команд над строкой, указанной в команде вставки.
  - Не просматривается при поиске по адресу.
  - Не изменяется другими командами редактирования.
  - Не изменяет счетчик строк.

Замена строк

Команда **s** заменяет символы в указанных строках файла ввода.

Если команда найдет набор символов, соответствующий регулярному выражению, заданному в параметре *Шаблон*, она заменит их другим набором символов, задаваемым параметром *Строка*.

Параметр *Строка* представляет собой последовательность символов (цифр, букв и знаков). *Строка* может содержать два специальных символа:

Символ	Использование
&	Этот символ из поля *строка* заменяется в строках ввода на символы, заданные в параметре *шаблон*. Пример:

```
s/boy/&s/
```

указывает программе **sed**, что нужно найти слово boy в файле ввода и скопировать этот шаблон в вывод, добавив к нему букву s. Следовательно:

Строка:
The boy look at the game.

Будет заменена на:
The boys look at the game.

Символ	Использование
\d	**d** - одна цифра. Этот символ в параметре *Строка* заменяется на символы строки ввода, совпадающие подстрокой *Шаблона* номер **d**. Эти подстроки начинаются и заканчиваются символом \. Например, команда: `s/\(stu\)\(dy\)/\1r\2/` **Строка:** The study chair **Будет заменена на:** The sturdy chair

Буквы, указываемые в качестве флагов, задают следующие параметры редактирования:

Символ	Использование
g	Заменяет в указанной строке (строках) все вхождения *Шаблона* на *Строку*. После вставки в символах *Строки* поиск *Шаблона* не выполняется. Например, команда: `s/r/R/g` выполняет следующее действие: **Строка:** the red round rock **Будет заменена на:** the Red Round Rock
p	Печатает (в STDOUT) строку, содержащую вхождение *Шаблона*.
w *файл*	Записывает в указанный файл строку, содержащую вхождение *шаблона*. Если файл *имя_файла* существует, он будет заменен; в противном случае он будет создан. В процессе редактирования можно задать до 10 различных файлов ввода или вывода. Между именем команды **w** и параметром *имя_файла* должен быть указан строго один пробел.

Общие библиотеки и общая память

Этот аргумент посвящен функциям операционной системы, предназначенным для работы с общими библиотеками и общей памятью.

В операционной системе предусмотрены средства создания и использования динамически подключаемых общих библиотек. Механизм динамического подключения позволяет загрузчику считывать во время выполнения внешние символы, определенные в общей библиотеке и упоминаемые в пользовательской программе.

Код общей библиотеки не хранится на диске в виде исполняемого образа. Общий код один раз загружается в сегмент памяти, отведенный для общей библиотеки, после чего с ним могут работать все процессы. Ниже перечислены преимущества общих библиотек:

- Экономия дискового пространства за счет того, что код общей библиотеки не включается в исполняемый код программ.
- Экономия памяти за счет того, что код общей библиотеки загружается в память только один раз.
- Возможное сокращение времени загрузки, если код общей библиотеки уже находится в памяти.
- Возможное повышение производительности за счет снижения количества страничных ошибок, если код общей библиотеки уже находится в памяти. Однако производительность снижается при вызове библиотечных функций, длина которых составляет от 1 до 8 инструкций.

Для того чтобы имена, определенные в общей библиотеке, были доступны для модулей, ссылающихся на них, эти имена необходимо явно экспортировать с помощью файла экспорта, за исключением случаев, когда применяются опции -bexpall. В первой строке файла экспорта можно указать полное имя общей библиотеки. В следующих строках указываются экспортируемые имена.

Информация, связанная с данной:

ar

as
dump
ipcs
ipcrm
id
pagesize
rtl_enable
update
vmstat
XCOFF

Общие объекты и динамическая компоновка

По умолчанию программы компонуются таким образом, что символы, импортируемые из общих объектов, связываются со своими определениями при загрузке программ.

Это происходит и в тех случаях, когда символ определен также внутри самой программы или другого общего объекта, необходимого для ее работы.

динамический компоновщик
: Общий объект, позволяющий динамически связывать символы в программах, скомпонованных особым образом.

Для того чтобы включить в программу динамический компоновщик, укажите при компоновке опцию **-brtl**. Если указана эта опция, выполняются следующие действия:

- В программу добавляется ссылка на динамический компоновщик. Процедура запуска такой программы (**/lib/crt0.o**) вызывает динамический компоновщик перед вызовом функции main.
- Все входные файлы, используемые в качестве общих объектов, указываются в разделе загрузчика как зависящие от вашей программы. Общие объекты указываются в том порядке, в котором они были перечислены в командной строке. Как следствие, системный загрузчик загружает все эти объекты вместе с программой, что позволяет динамическому компоновщику брать определения из них. Если в командной строке не указана опция **-brtl**, то в списке зависимых объектов будут присутствовать только те объекты, которые явно указаны в программе. Общие объекты, не указанные в программе, не будут добавлены в список зависимых объектов, даже если в них хранятся определения, необходимые для других общих объектов, используемых программой.
- Общие объекты, хранящиеся в архиве, заносятся в список зависимых объектов только в том случае, если для архива указана опция автоматической загрузки элемента, в котором хранятся общие объекты. Например, для автоматической загрузки элемента архива **foo.o** нужно создать файл, содержащий следующие строки:

  ```
  # autoload
  #! (foo.o)
  ```

  и добавить его в архив в качестве отдельного элемента.
- При компоновке в динамическом режиме с флагом **-l** могут указываться файлы с расширением **.so** и **.a**. Следовательно, если вы укажете опцию **-lfoo**, и при этом существуют файлы **libfoo.so** и **libfoo.a**, то будет использоваться тот из них, который будет найден первым. По умолчанию компоновка выполняется в динамическом режиме. Для выключения динамического режима нужно указать опцию **-bstatic**.

Динамический компоновщик работает практически по тому же алгоритму, что и команда **ld**. Единственное отличие заключается в том, что могут обрабатываться только экспортированные символы. Даже при динамической компоновке у системного загрузчика должна быть возможность найти и обработать все символы, присутствующие как в главной программе, так и во всех зависящих от нее модулях. В связи с этим, если вы удалите из модуля определение, ссылка на которое есть в главной программе, то программа не будет выполнена, даже если в другом модуле есть другое определение этого символа.

Динамический компоновщик может обработать все ссылки на символы, импортированные из других модулей. Ссылки на символы, определенные в том же модуле, обрабатываются только в том случае, если эти символы определены с опцией динамической компоновки.

Начиная с AIX 4.2, в большинстве общих модулей, поставляемых с AIX, задана опция динамической компоновки большинства экспортируемых переменных. Динамическая компоновка функций возможна только в случае вызова функции через указатель. Например, в текущей версии AIX вызовы процедуры **malloc** из общего объекта **shr.o** (библиотека **/lib/libc.a**) нельзя перенаправить для вызова другой процедуры, даже если определение **malloc** есть в функции main или в другом общем модуле. Большинство поставляемых с системой общих модулей можно перекомпоновать с опцией динамической компоновки для функций и переменных. Для этого нужно воспользоваться командой **rtl_enable**.

Операции, выполняемые динамическим компоновщиком

Главная программа загружается и обрабатывается системным загрузчиком в обычном режиме. Если по каким-либо причинам главную программу загрузить не удается, процедура **exec()** завершается с ошибкой и динамический компоновщик не вызывается. При успешной загрузке главной программы управление передается динамическому компоновщику, который обрабатывает символы по схеме, описанной ниже. После завершения работы динамического компоновщика вызываются процедуры инициализации (если они предусмотрены), после чего вызывается функция main.

Динамический компоновщик обрабатывает модули в порядке прямой зависимости - начинает с главной программы, затем переходит к первому непосредственно зависящему от нее модулю, затем к первому зависящему от него и т.д. При поиске определения символа модули просматриваются в том же порядке. Кроме двух исключений описанных ниже, всегда применяется первое найденное определение. Если первое найденное определение относится к неопределенному отложенному символу, то символ считается неопределенным и обработка прекращается. Если первое найденное определение будет помечено как BSS (т.е. как символ типа **XTY_CM**, соответствующий неинициализированной переменной), то динамический компоновщик попытается найти другое определение, не помеченное как BSS и не являющееся неопределенным отложенным символом. Если такое определение будет найдено, то будет использоваться оно. В противном случае будет применяться первое найденное определение.

В разделе загрузки каждого модуля перечислены импортируемые символы, которые обычно бывают определены в другом модуле, а также экспортируемые символы, которые обычно определяются в данном модуле. Символы, которые импортируются и экспортируются одновременно, называются внешними. С такими символами можно работать так, как будто они определены в данном модуле, хотя на самом деле они определены в другом модуле.

Некоторые символы могут быть помечены как "отложенные". Такие символы не обрабатываются динамическим компоновщиком. Они обрабатываются системным загрузчиком либо с помощью процедуры **loadbind()**, либо путем явной загрузки дополнительного модуля с помощью процедуры **load()** или **dlopen()**.

Ссылки на все импортируемые символы, за исключением отложенных, всегда могут быть обработаны повторно. К моменту запуска динамического компоновщика системный загрузчик уже обработает большинство символов. Ссылки на все импортируемые символы обрабатываются в соответствии со своим определяющим экземпляром. Если определяющего экземпляра символа не существует, то в поток stderr направляется сообщение об ошибке. Если в ссылке на символ и в определении символа не совпадают строки проверки типов, то также выдается сообщение об ошибке.

Ссылки на экспортируемые символы, расположенные в таблице размещения в разделе загрузчика, также обрабатываются в соответствии со своими определяющими экземплярами. (Внешние символы обрабатываются вместе с прочими импортируемыми символами по схеме, описанной выше.) В зависимости от способа компоновки модуля, некоторые ссылки на экспортируемые символы могут обрабатываться во время компоновки программы, без возможности повторной обработки. Поскольку экспортируемые символы определены в текущем модуле, для них всегда будет существовать определяющий экземпляр (кроме случая, когда первый найденный экземпляр будет определением отложенного символа). Поэтому при

повторной обработке экспортируемых символов ошибки хотя и встречаются, но крайне редко. Как и при обработке импортируемых символов, при несовпадении строк проверки типов выдается сообщение об ошибке.

После обработки символа модуль, в котором он используется, помечается как зависящий от модуля, в котором этот символ определен. Это позволяет избежать преждевременного удаления модуля с определением символа из адресного пространства. Это важный этап, потому что модули, загруженные процедурой **dlopen** и содержащие определения символов, применяемых в других модулях, могли бы быть по ошибке закрыты процедурой **dlclose**.

В таблице символов в разделе загрузчика нет никакой информации о выравнивании и длине символов. В связи с этим ошибки, связанные с длиной и выравниванием символов, не выявляются на этапе обработки символов. Такие ошибки обнаруживаются только при выполнении программы.

Если при компоновке возникли какие-либо ошибки, то после обработки всех модулей динамический компоновщик вызывает функцию **exit** и передает ей код возврата 144 (0x90). В противном случае вызываются функции инициализации или функция **main()**.

Создание общего объекта с поддержкой динамической компоновки

Для создания общего объекта с поддержкой динамической компоновки нужно указать флаг **-G** в командной строке компоновщика. В этом случае при компоновке объекта выполняются следующие действия:

1. Экспортируемым символам присваивается атрибут `nosymbolic`, что в дальнейшем позволяет динамическому компоновщику обрабатывать эти символы.
2. Допускается применение неопределенных символов (см. описание опции **-berok**). Такие символы помечаются как импортируемые из символьного модуля "..". Символы, импортируемые из модуля "..", должны обрабатываться динамическим компоновщиком, потому что системный загрузчик не может их обработать.
3. Выходному файлу присваивается тип модуля `SRE` (как если бы в командной строке была указана опция **-bM:SRE**).
4. Все общие объекты, указанные в командной строке, будут помечены как зависящие от выходного модуля (как и при компоновке программы с опцией **-brtl**).
5. Общие объекты из архива указываются в командной строке в том случае, если для них задан атрибут `autoload`.

Флаг **-G** по умолчанию включает опцию **-berok**. Эта опция скрывает ошибки, которые могут быть обнаружены во время компоновки. Если вы хотите, чтобы все используемые символы были определены к моменту компоновки модуля, укажите с флагом **-G** опцию **-bernotok**. В этом случае будут выданы сообщения об ошибках для всех неопределенных символов.

Общие библиотеки и частичная загрузка

По умолчанию при загрузке модуля системный загрузчик автоматически загружает и все зависимые модули. Это объясняется тем, что при компоновке модуля в его загрузочном разделе сохраняется список зависимых модулей.

Модуль	Описание
dump -H	Команда просмотра списка зависимых модулей.
-blazy	В AIX 4.2.1 и в последующих версиях - опция компоновщика, указывающая, что при компоновке модуля следует загрузить только часть зависимых модулей, если функция из этого модуля вызывается в первый раз.

Частичная загрузка позволяет повысить производительность программы, если большая часть зависимых модулей в действительности не используется. С другой стороны, при каждом вызове функции из модуля, загруженного таким образом, выполняется около 7 дополнительных команд, и при первом вызове функции необходимо загрузить модуль с определениями и внести изменения в вызов функции. Поэтому не рекомендуется применять частичную загрузку в случае, если модуль вызывает функции, содержащиеся в большинстве зависимых модулей.

При первом вызове функции, определенной в частично загруженном модуле, система пытается загрузить модуль с определениями и найти в нем нужную функцию. Если этот модуль не найден или если он не экспортирует данную функцию, по умолчанию выдается стандартное сообщение об ошибке, и программа завершает работу с кодом возврата 1. В приложении можно реализовать собственный обработчик ошибок путем вызова функции **_lazySetErrorHandler** с адресом пользовательского обработчика ошибок в качестве параметра. При вызове обработчика ошибок в него передаются три аргумента: имя модуля, имя символа и значение, обозначающее причину ошибки. Если обработчик ошибок возвращает значение, он должен возвращать адрес функции, употребляемой вместо данной. Функция **_lazySetErrorHandler** возвращает либо адрес обработчика ошибок, либо NULL, если обработчика нет.

Применение частичной загрузки обычно не приводит к изменениям в работе программы, за несколькими важными исключениями. Во-первых, может нарушиться работа программ, зависящих от порядка загрузки модулей, так как при частичной загрузке модули могут загружаться в другом порядке, а некоторые - не загружаться вообще.

Во-вторых, могут возникнуть ошибки в работе программ, в которых выполняется сравнение указателей на функции, поскольку у одной и той же функции может быть несколько адресов. Например, если модуль A вызывает функцию `f' из модуля B, а при компоновке модуля A указана частичная загрузка модуля B, то адрес функции `f', вычисленный в модуле A, будет отличаться от адреса функции `f', вычисленного в других модулях. Следовательно, при применении частичной загрузки два указателя на одну и ту же функцию могут быть различными.

В-третьих, если какие-либо модули загружаются по относительным именам файлов, а программа изменяет рабочие каталоги, то программа может не найти зависимые модули, когда они потребуются. Если применяется частичная загрузка, то для зависимых модулей на этапе компоновки следует указывать только полные имена файлов.

Решение о том, следует ли применять частичную загрузку, принимается на этапе компоновки отдельно для каждого модуля. В одной и той же программе некоторые модули можно загружать полностью, а некоторые - частично. Если на этапе компоновки любого модуля в нем обнаруживается ссылка на переменную из зависимого модуля, то этот модуль будет загружен полностью. Если все ссылки на модуль являются ссылками на символы функций, то зависимый модуль может быть загружен частично.

Частичную загрузку можно применять как в приложениях с нитями, так и в приложениях без нитей.

Трассировка частичной загрузки

Специальная процедура времени выполнения позволяет обнаружить загрузку модулей на этапе выполнения. Эта процедура изменяет значение переменной среды **LDLAZYDEBUG**. Последнее представляет собой десятичное, восьмеричное (с префиксом 0) или шестнадцатеричное (с префиксом 0x) число, равное сумме нескольких из перечисленных ниже значений или одному из них:

Переменная	Описание
1	Наличие ошибок загрузки или поиска. Если нужный модуль не найден, то выдается сообщение и вызывается обработчик ошибок для частичной загрузки. Если модуль найден и загружен, но требуемый символ в нем недоступен, то выдается сообщение и после этого вызывается обработчик ошибок.
2	Запись сообщений трассировки в поток **stderr** вместо **stdout**. По умолчанию эти сообщения помещаются в стандартный поток вывода. Это значение позволяет помещать их в стандартный поток сообщений об ошибках.
4	Вывод имени загружаемого модуля. Если для вызова функции необходимо загрузить новый модуль, то после обнаружения и загрузки этого модуля на экране появляется его имя. Это происходит только при обработке первой ссылки на функцию из этого модуля, т.е. после загрузки модуля он остается доступным для последующих ссылок на функции из него. Дополнительные операции загрузки не нужны.
8	Вывод имени вызываемой функции. На экран выводится имя запрошенной функции вместе с именем модуля, в котором она должна содержаться. Эта информация выводится до загрузки модуля.

Именованные области общей библиотеки

По умолчанию, процессы в AIX работают с общими библиотеками, используя глобальный набор сегментов, который называется глобальной областью общей библиотеки.

Для 32-разрядных процессов эта область состоит из одного сегмента для текста общей библиотеки (сегмент 0xD) и одного сегмента для предварительно перемещенных данных библиотеки (сегмент 0xF). Использование общего текста и предварительное перемещение данных увеличивает производительность в системах с большим числом процессов, работающих с общими библиотеками.

Поскольку глобальная область общей библиотеки является одиночным ресурсом фиксированного размера, попытки совместного использования набора библиотек, размер которого превышает емкость этой области, будут неудачными. В этом случае часть библиотек процессов загружается частным образом. Загрузка библиотек частным образом, в противоположность общим библиотекам, использует частное адресное пространство в процессе и требует большего пространства подкачки, что приводит к снижению производительности системы в целом.

Для того, чтобы обойти данное ограничение глобальной области общей библиотеки, AIX 5.3 поддерживает *именованные области общей библиотеки*, которые имеют следующие преимущества:

- Именованная область общей библиотеки замещает глобальную область общей библиотеки для группы процессов.
- Именованная область общей библиотеки предоставляет группе процессов полный объем общей библиотеки в том же расположении в эффективном адресном пространстве, что и глобальная область общей библиотеки (сегменты 0xD и 0xF).
- Функцию именованной области общей библиотеки можно включить с помощью переменной среды **LDR_CNTRL**. При этом не требуется вносить какие-либо изменения в существующий двоичный код.
- В системе могут быть одновременно активны несколько именованных областей общей библиотеки.
- Процессы указывают определенную именованную область общей библиотеки с помощью уникального имени. Это имя выбирается процессом, вызвавшим создание области.
- Именованные области общей библиотеки доступны только для 32-разрядных процессов.

Поскольку использование конкретной именованной области общей библиотеки ограничено процессами, которые ее запрашивают, пространство этой области не будет использовано процессами, работающими с глобальной областью общей библиотеки area или с другой именованной областью. Это снижение конкуренции за пространство в именованной области общей билиотеки предоставляет преимущество для

процессов, использующих эту область. Эти процессы потребляют меньше частного адресного пространства и имеют больше возможности для совместного использования библиотек. Использование именованной области общей библиотеки процессами, которые работают с общими библиотеками, позволяет оптимизировать использование адресного пространства процессов и уменьшить требования к пространству подкачки, что повышает производительность системы в целом.

Альтернативная модель памяти (doubletext32)

В дополнение к модели по умолчанию для памяти области общей библиотеки (один сегмент выделен для текста общей библиотеки и один сегмент выделен для предварительно перемещенных данных библиотеки) именованные области общей библиотеки поддерживают альтернативную модель памяти, в которой оба сегмента выделяются для текста общей библиотеки. Эта модель полезна для групп процессов, совместно использующих более 256 Мб текста библиотеки. Обратите внимание, что поскольку в этой альтернативной модели памяти не выполняется предварительное перемещение данных библиотеки, это может привести к некоторому снижению производительности во время загрузки модулей (как для зависимостей во время выполнения, так и динамически загружаемых модулей). Поэтому фактическое увеличение производительности при увеличенной емкости текста общей библиотеки следует рассматривать отдельно для каждого случая.

Interface

Права доступа

Процесс запрашивает использование именованной области общей библиотеки с помощью переменной среды **LDR_CNTRL** с опцией **NAMEDSHLIB** в своей среде во время выполнения. Новая опция имеет следующий синтаксис:

```
NAMEDSHLIB=имя[,атрибут][,атрибут2]...[,атрибутN]
```

Допустимое имя может быть любой строкой, соответствующей регулярному выражению, `[A-Za-z0-9_\.]+` (содержащая только алфавитно-цифровые символы, символы подчеркивания и точки).

Допустимая строка имени должна завершаться одним из следующих символов:

- `@` (собачка): Ограничитель для нескольких опций **LDR_CNTRL**
- `,` (запятая): Ограничитель для атрибутов **NAMEDSHLIB**
- `\0` (нуль): Терминатор строки среды **LDR_CNTRL**

Если указана недопустимая строка имени, вся опция **NAMEDSHLIB** игнорируется. Если задан недопустимый атрибут, то игнорнируется только этот атрибут. В настоящее время поддерживается только один атрибут: **doubletext32**.

Не существует ограничений по правам доступа на использование именованных областей общей библиотеки. Все запросы на использование области удовлетворяются.

Создание

Не существует специального интерфейса для создания именованной области общей библиотеки. Когда процесс запрашивает использование не существующей именованной области общей библиотеки, такая область создается автоматически.

Очистка

Система удаляет неиспользуемые библиотеки из именованной области общих библиотек с помощью тех же механизмов, которые применяются для глобальной области общих библиотек:

- Автоматическое удаление неиспользуемых библиотек выполняется при заполнении области.
- Принудительное удаление неиспользуемых библиотек можно выполнить с помощью команды **slibclean**.

Уничтожение

Не существует специального интерфейса для уничтожения именованной области общей библиотеки. После выхода последнего процесса, использующего именованную область общих библиотек (значение usecount области становится равным нулю), область автоматически уничтожается.

Атрибуты

Атрибуты **NAMEDSHLIB** проверяются программой загрузки системы только во время создания именованной области общих библиотек. Поэтому в запросах на использование существующей именованной области общих библиотек не обязательно должны быть точно указаны атрибуты, соответствующие атрибутам, заданным при создании (сбой запроса вследствие несовпадения атрибутов не происходит). Однако поскольку система автоматически разрушает неиспользуемые именованные области общих библиотек, рекомендуется всегда указывать атрибуты, даже в случае запроса на использование существующей именованной области.

Примеры

1. Запускается пара приложений, использующих именованную область общих библиотек с именем *XYZ*, в которой один сегмент выделен для текста общей библиотеки и один сегмент выделен для предварительно перемещенных данных библиотеки. Для этого используются следующие команды:

   ```
   $ export LDR_CNTRL=NAMEDSHLIB=XYZ
   $ xyz_app
   $ xyz_app2
   ```

2. Запускается пара приложений, использующих именованную область общих библиотек с именем *more_shtext*, в которой оба сегмента выделены для текста общих библиотек. Для этого используются следующие команды:

   ```
   $ export LDR_CNTRL=NAMEDSHLIB=more_shtext,doubletext32
   $ mybigapp
   $ mybigapp2
   ```

Создание общей библиотеки

В этом разделе приведены инструкции по созданию общей библиотеки.

Предварительные действия

1. Создайте один или несколько исходных файлов, которые будут откомпилированы и скомпонованы в общую библиотеку. Эти файлы будут содержать экспортированные символы, на которые могут ссылаться другие исходные файлы.

   В примерах этого раздела используются два исходных файла, `share1.c` и `share2.c`. Файл `share1.c` содержит следующий текст:

   ```
   /************
     * share1.c: исходный код общей библиотеки.
   *************/

   #include <stdio.h>

   void func1 ()
    {
         printf("func1 called\n");
    }

   void func2 ()
    {
         printf("func2 called\n");
    }
   ```

   Файл `share2.c`содержит следующий текст:

```
/************
  * share2.c: исходный код общей библиотеки.
*************/

void func3 ()
 {
      printf("func3 called\n");
 }
```

Экспортируемыми символами в этих файлах являются func1, func2 и func3.

2. Создайте главный исходный файл, ссылающийся на экспортируемые символы, которые будут входить в общую библиотеку.

   В примерах этого раздела используется главный исходный файл main.c. Файл main.c содержит следующий текст:

```
/************
  * main.c: содержит ссылки на символы, заданные в
 * файлах share1.c и share2.c
*************/

#include <stdio.h>

    extern void func1 (),
                func2 (),
                func3 ();
main ()
 {
                func1 ();
                func2 ();
                func3 ();
 }
```

3. Создайте файл экспорта, необходимый для явного экспорта символов общей библиотеки, которые могут использоваться во внешних объектных модулях.

   В примерах этого раздела используется файл экспорта shrsub.exp. Файл shrsub.exp содержит следующий текст:

```
#! /home/sharelib/shrsub.o
* Полный путь к файлу объекта общей библиотеки
func1
func2
func3
```

   Символы #! применяется только при использовании файла для импорта. В этом случае строка с #! указывает имя файла общей библиотеки, который будет применяться на этапе выполнения.

Процедура

1. Скомпилируйте и скомпонуйте два исходный файла, содержащих код общей библиотеки. (Предполагается, что текущий каталог - **/home/sharedlib**.) Для компиляции и компоновки исходных файлов введите следующие команды:

```
cc -c share1.c
cc -c share2.c
cc -o shrsub.o share1.o share2.o -bE:shrsub.exp -bM:SRE -bnoentry
```

   В результате выполнения этих команд будет создана общая библиотека shrsub.o в каталоге **/home/sharedlib**.

   Флаг -bM:SRE

   Помечает результирующий объектный файл shrsub.o как реентерабельную объектную библиотеку

   Каждый процесс, который применяет общий код, получает индивидуальную копию данных, размещенную в защищенной области процесса.

флаг
Устанавливает фиктивную точку входа `_nostart` для переопределения точки входа по умолчанию `_start`

Флаг -bnoentry
Сообщает редактору связей о том, что общая библиотека не имеет точки входа.

В общей библиотеке может присутствовать точка входа, однако системный загрузчик не использует ее при загрузке общей библиотеки.

2. Для помещения общей библиотеки в архивный файл введите следующую команду:
```
ar qv libsub.a shrsub.o
```

Это необязательная операция. Однако помещение библиотеки в архивный файл упрощает ее применение при компоновке программ, так как для архивных библиотек можно использовать ключи **-l** и **-L** команды **ld**.

3. Откомпилируйте и скомпонуйте главный исходный файл для создания исполняемого файла. (Предполагается, что в текущем каталоге присутствует файл **main.c**.) Для этого введите следующую команду:
```
cc -o main main.c -lsub -L/home/sharedlib
```

Если общая библиотека не была помещена в архив, введите команду:
```
cc -o main main.c /home/sharedlib/shrsub.o -L/home/sharedlib
```

После этого программу `main` можно запускать. Символы `func1`, `func2` и `func3` помечены для отложенного связывания на этапе загрузки. На этапе выполнения системный загрузчик загрузит модуль общей библиотеки (если он еще не загружен) и динамически обработает обнаруженные ссылки.

Примечание: При создании общей библиотеки объекта C++ в файле экспорта следует использовать измененные имена символов C++. Обратите внимание, что в компиляторе C++ должна быть предусмотрена опция создания общей библиотеки. Дополнительная информация приведена в документации по компилятору.

Флаг -L
Добавляет указанный каталог (в данном случае, `/home/sharedlib`) в путь поиска библиотеки, хранящийся в разделе загрузчика программы.

Во время работы программы путь поиска библиотек указывает загрузчику, где расположены общие библиотеки.

Переменная среды LIBPATH
Может использоваться в качестве пути поиска библиотек; содержит список каталогов, перечисленных через точку с запятой. Формат этой переменной совпадает с форматом переменной **PATH**.

Каталоги из этого списка применяются при обработке ссылок на внешние объекты. Каталоги **/usr/lib** и **/lib** содержат общие библиотеки и, как правило, должны присутствовать в пути поиска библиотек.

Адресное пространство программы - обзор

В состав базовой операционной системы включен набор служб распределения памяти между приложениями.

В их число входят средства выделения памяти, отображения файлов в памяти и контроля за использованием памяти приложениями. Помимо описания этих служб, в данном разделе приведена информация об архитектуре и стратегии управления памятью в системе.

Архитектура памяти системы - введение

Схема работы средств управления памятью позволяет расширить возможности оборудования с помощью программных средств. Из-за отсутствия однозначного соответствия между адресным пространством и физической памятью адресное пространство обычно называется виртуальной памятью.

Подсистемы ядра и аппаратное обеспечение, осуществляющее преобразование виртуальных адресов в физические, составляют подсистему управления памятью. Действия ядра, обеспечивающие правильное совместное использование оперативной памяти, реализуют стратегию управления памятью. В следующих разделах приведено более подробное описание подсистемы управления памятью.

Пространство физических адресов в 64-разрядных системах

Аппаратное обеспечение предоставляет доступ к адресам виртуальной памяти, лежащим в непрерывном диапазоне от `0x00000000000000000000` до `0xFFFFFFFFFFFFFFFFFFFF`. Общее адресное пространство, таким образом, составляет более 1000000000000 Тб. Команды обращения к памяти оперируют с 64-разрядными адресами: 36 бит указывают сегментный регистр, а 28 бит задают смещение внутри сегмента. Такая схема адресации обеспечивает доступ более чем к 64 миллионам сегментов, до 256 Мб каждый. Каждый сегментный регистр содержит 52-разрядный ИД сегмента, который вместе с 28-разрядным смещением образует адрес виртуальной памяти. Этот 80-разрядный виртуальный адрес ссылается на единое системное пространство виртуальной памяти.

Адресное пространство процесса 64-разрядное, то есть программы используют 64-разрядные указатели. При этом процессы или обработчики прерываний могут обращаться только к тем областям системной виртуальной памяти (сегментам), ИД которых хранятся в сегментном регистре.

Адресация сегментных регистров

Ядро системы загружает несколько сегментных регистров, тем самым неявно разрешая всем процессам обращение к областям памяти, которые нужны для работы большинства процессов. Эти регистры позволяют обращаться к двум сегментам ядра, сегменту общих библиотек и к сегменту устройств ввода-вывода, совместно используемым всеми процессами. Программам, не входящим в ядро, содержимое этих сегментов доступно только для чтения. Существует также сегмент для системного вызова процесса **exec**, который совместно используется (только для чтения) и несколькими процессами, применяемыми в одной программе; частный сегмент данных общих библиотек, содержащий данные библиотек, доступные только для чтения; сегмент, доступный для чтения-записи, и частный сегмент процесса. В остальные сегменты можно загрузить значения в соответствии с принципам отображения памяти для увеличения общего объема памяти, либо для работы с файлами, отображаемыми в память, в соответствии с правилами, устанавливаемыми ядром.

32-разрядная адресация и косвенный доступ к данным позволяют реализовать в системе интерфейс, не зависящий от фактического объема системной виртуальной памяти. Некоторые сегментные регистры используются совместно всеми процессами, другие - только ограниченным набором процессов, некоторые же могут использоваться только одним процессом. Для того чтобы несколько процессов могли использовать сегментный регистр совместно, они должны загрузить один и тот же ИД сегмента.

Пространство подкачки

Для организации большого виртуального пространства при ограниченном объеме физической памяти система использует физическую память в качестве рабочего пространства и хранит неактивные данные и программы на диске. Область диска, содержащая эти данные, называется пространством подкачки. Объем виртуальной памяти измеряется в страницах. Одна страница содержит 4 Кб данных, которые можно перемещать из физической памяти во вспомогательную и наоборот. Если системе потребовались данные или программа из пространства подкачки, она выполняет следующие действия:

1. Находит неактивную в данный момент область памяти.
2. Обеспечивает, чтобы последняя копия данных или программы из этой области памяти была записана в пространство подкачки на диске.

3. Считывает нужную программу или данные из пространства подкачки на диске и помещает их в освобожденную область памяти.

Стратегия управления памятью

Функция преобразования реальных адресов в виртуальные и большинство других функций работы с виртуальной памятью выполняются в системе Диспетчером виртуальной памяти (VMM). VMM реализует стратегию управления виртуальной памятью, допускающую создание сегментов, размер которых превышает общий объем доступной физической памяти. С этой целью VMM создает список свободных страниц физической памяти, используемый при загрузке страниц.

VMM должен периодически добавлять страницы в этот список, удаляя данные некоторых страниц из физической памяти. Процесс перемещения данных между памятью и диском называется "подкачкой". VMM выполняет подкачку по алгоритму захвата страниц, разделяющему страницы на три класса с различными критериями входа и выхода:

- страницы рабочей памяти
- страницы локальных файлов
- страницы удаленных файлов

Рабочие страницы имеют максимальный приоритет, страницы локальных файлов - промежуточный, и страницы удаленных файлов - самый низкий приоритет.

Кроме того, для выбора заменяемых страниц VMM использует технологию, называемую "часовым алгоритмом". В соответствии с этим алгоритмом для каждой страницы устанавливается бит обращения, позволяющий выбирать страницы, которые недавно использовались системой. При вызове процедура захвата страниц циклически просматривает таблицу страниц и проверяет биты обращения к каждой странице. Если обращение к странице не выполнялось и она не закреплена (и отвечает еще некоторым специальным условиям), то эта страница будет захвачена и помещена в список свободных страниц. Страницы, которым были обращения, нельзя захватить, но можно сбросить их бит обращения, в результате чего при следующем запуске процедуры захвата они будут считаться неактивными.

Выделение памяти

В версии 3 данной операционной системы выделение памяти приложениям осуществляется по принципам отложенной подкачки. Это означает, что когда память выделяется приложению некоторой процедурой (например, **malloc**), пространство подкачки для этой памяти не будет выделено до тех пор, пока приложение не обратится к ней.

Понятия, связанные с данным:

“Отображение памяти - основные сведения”
Скорость обработки команд приложения в системе обратно пропорциональна числу операций считывания данных, хранящихся за пределами адресуемой программами памяти.

“Требования программ к пространству подкачки” на стр. 612
Объем пространства подкачки, который требуется приложению, зависит от типа действий, выполняемых в системе. При нехватке пространства подкачки возможно аварийное завершение работы процессов.

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Отображение памяти - основные сведения

Скорость обработки команд приложения в системе обратно пропорциональна числу операций считывания данных, хранящихся за пределами адресуемой программами памяти.

Существует два способа уменьшения числа транзакций, связанных с внешними операциями чтения/записи. Можно отобразить данные из файлов в адресное пространство процесса, а можно отобразить процессы в неименованные области памяти, которые могут совместно использоваться несколькими процессами.

Отображение файлов в память позволяет обеспечить доступ к файлам путем прямого помещения их содержимого в адресное пространство процесса. Отображенные файлы позволяют значительно снизить объем ввода-вывода, поскольку данные из этих файлов не нужно копировать в буферы данных процесса, как это делают процедуры **read** и **write**. Если один и тот же файл отображен несколькими процессами, то они совместно используют соответствующую область памяти, применяя при этом средства синхронизации и связи.

Отображенные области памяти, называемые также общими областями памяти, могут служить большими пулами для обмена данными между процессами. Стандартные процедуры не реализуют блокировки и не позволяют управлять доступом к общим данным на уровне процессов. Следовательно, в процессах, работающих с общей областью памяти, для обеспечения целостности данных и предотвращения конфликтов, связанных с доступом, должен применяться сигнальный или семафорный способ управления доступом. Преимущества общих областей памяти особенно ярко проявляются в случаях, когда объем данных, которыми обмениваются процессы, очень велик для передачи их с помощью сообщений, или при работе нескольких процессов с общей базой данных большого объема.

В системе есть два способа отображения файлов и неименованных областей памяти. Как правило, для создания общих сегментов памяти и работы с ними служит следующий набор процедур, называемый службами **shmat**:

Функция	Определение
shmctl	Управляет операциями с общими областями памяти
shmget	Получает или создает общий сегмент памяти
shmat	Подключает сегмент общей памяти к процессу. Не позволяет создавать отображения блочных устройств.
shmdt	Отключает общий сегмент памяти процесса
mprotect	Изменяет права доступа указанного диапазона адресов сегмента общей памяти.
disclaim	Удаляет отображенные данные в указанном диапазоне адресов общего сегмента памяти

Функция **ftok** генерирует ключ, с помощью которого функция **shmget** создает общий сегмент

Второй набор служб, который называется **mmap**, обычно применяется для работы с отображенными файлами, хотя с помощью этих функций можно создавать и общие сегменты памяти.

Все операции, допустимые для блоков памяти, полученных в результате выполнения операции mmap() над файлом, допустимы для блоков памяти, полученных в результате выполнения операции mmap() над блочным устройством. Блочное устройство - это особый файл, обеспечивающий доступ к драйверу устройства с блочным интерфейсом. Блочный интерфейс - это интерфейс доступа к драйверу устройства, позволяющий оперировать блоками данных фиксированного размера. Как правило, такие интерфейсы применяются для работы с накопителями.

В этот набор входят следующие процедуры:

Функция	Определение
madvise	Сообщает системе о предположительном объеме подкачки для процесса
mincore	Определяет расположение страниц памяти
mmap	Отображает объектный файл в виртуальную память. Позволяет отображать блочные устройства в отдельных процессах.
mprotect	Изменяет ограничения доступа, связанные с отображением памяти
msync	Синхронизирует отображенный файл с запоминающим устройством, на котором от расположен
munmap	Освобождает область памяти, выделенную для отображения

Функции **msem_init** , **msem_lock**, **msem_unlock**, **msem_remove**, **msleep** и **mwakeup** обеспечивают управление доступом для процессов, отображаемых с помощью служб **mmap**.

Более подробно процедура отображения в память описана в следующих разделах:

Сравнение mmap и shmat

Как и для служб shmat, размер области адресного пространства, доступной для отображения файлов с помощью служб mmap, зависит от того, является ли соответствующий процесс 32- или 64-разрядным. В 32-разрядных процессах для отображения доступен диапазон адресов 0x30000000-0xCFFFFFFF, общий объем этой области - 2,5 Гб. Для отображения файлов доступно адресное пространство с адресами из диапазонов 0x30000000-0xCFFFFFFF и 0xE0000000-0xEFFFFFFF, общий объем - 2,75 Гб. В AIX 5.2 и более поздних версий с 32-разрядными процессами, использующими модель сверхбольшого адресного пространства, для отображения доступен диапазон адресов 0x30000000-0xFFFFFFFF с общим объемом адресного пространства до 3,25 Гб.

Все диапазоны адресов, доступные для 32-разрядных приложений, подходят как для статического, так и для динамического отображения. Статическое отображение выполняется в случае, когда в приложении указано, что отображение должно находиться в фиксированной области адресного пространства. Динамическое отображение выполняется, если в приложении указано, что расположение отображения устанавливается системой.

Для 64-разрядных процессов, отображения **mmap** и **shmat** могут размещаться в двух областях адресного пространства. В первой области, с диапазоном адресов 0x07000000_00000000-0x07FFFFFF_FFFFFFFF, можно размещать как статические, так и динамическое отображения. Во второй области, с диапазоном адресов 0x30000000-0xCFFFFFFF, 0xE0000000-0xEFFFFFFF и 0x10_00000000-0x06FFFFFF_FFFFFFFF, можно размещать только статические отображения. Последний фрагмент этой области, с адресами в диапазоне 0x10_00000000-0x06FFFFFF_FFFFFFFF, используется также системным загрузчиком для размещения текста программы, статических данных и кучи, поэтому размещать статические отображения можно только в незанятых участках этого фрагмента.

Как **mmap**, так и **shmat** могут отобразить объект в память так, что он будет доступен нескольким процессам. Однако в процедуре **mmap**, в отличие от процедуры **shmat**, число таких отображений для одного объекта не ограниченно. Поскольку при этом создается несколько отображений одного и того же файла или сегмента памяти, то в приложениях, в которых несколько процессов помещают одни и те же отображенные файлы в свое адресное пространство, это может привести к снижению производительности.

Процедура **mmap** присваивает уникальный адрес объекта каждому процессу, отображающему этот объект. Программно это реализуется путем присвоения каждому процессу виртуального адреса, или псевдонима. Процедура **shmat** позволяет процессам использовать общий адрес отображенного объекта.

Поскольку в любой момент времени только один из псевдонимов страницы объекта можно преобразовывать в фактический адрес, только одно отображение **mmap** может ссылаться на эту страницу, не вызывая страничной ошибки. При любой ссылке на эту страницу из другого отображения (т.е. под другим псевдонимом) результатом будет страничная ошибка, вследствие которой существующее преобразование псевдонима в реальный адрес для этой страницы будет уничтожено. Для восстановления потребуется создать новое преобразование под другим псевдонимом. Процессы совместно используют страницы, перемещая их из одного преобразования в другое.

В приложениях, в которых несколько процессов помещают одни и те же отображенные файлы в свое адресное пространство, описанный процесс перехода может вызвать снижение производительности. В этом случае для повышения эффективности отображения файлов следует пользоваться процедурой **shmat**.

Примечание: В системах с процессором PowerPC одному и тому же физическому адресу может соответствовать несколько виртуальных адресов. Реальному адресу можно присваивать несколько псевдонимов - виртуальных адресов для различных процессов, при этом переключение адресов не требуется. Поскольку переключение адресов не происходит, производительность не снижается.

Службы **shmat** следует применять в следующих случаях:

- В 32-разрядных приложениях, если одновременно отображается менее 12 файлов, каждый из которых меньше 256 Мб.

- При отображении файлов объемом более 256 Мб.
- При отображении общих областей памяти, которые должны совместно использоваться процессами, не состоящими в отношениях "предок-потомок".
- При отображении файлов целиком.

Службы **mmap** следует применять в следующих случаях:

- Если важна переносимость приложения.
- При одновременном отображении большого числа файлов.
- При отображении фрагментов файлов.
- Если необходима защита отображенных данных на уровне страниц.
- Если необходимо создать закрытые отображения.

Для 32-разрядных приложений с ограниченным адресным пространством применяются "расширенные возможности **shmat**". Если установлено значение переменной среды **EXTSHM=ON**, то процессы, выполняемые в этой среде, могут создавать и подключать более одиннадцати общих сегментов памяти. Процесс может размещать такие сегменты в областях адресного пространства, размер которых совпадает с размером сегмента. В той же области памяти размером 256 Мб можно разместить другой сегмент, который будет начинаться сразу после окончания первого сегмента. Адрес, по которому процесс может размещать сегменты, должен совпадать с границей страницы, т.е. должен быть кратным **SHMLBA_EXTSHM** байт.

Для применения расширенных функций **shmat** есть некоторые ограничения. Выделяемые общие области памяти не должны служить буферами ввода-вывода, которые обработчик прерываний выгружает из памяти. Те же ограничения относятся и к буферам ввода-вывода **mmap**.

С помощью этой переменной среды можно либо отвести для приложения более 11 сегментов, если **EXTSHM=ON**, либо повысить производительность обращения к отведенным 11 (или менее) сегментам, если значение переменной равно OFF. Еще раз подчеркнем, что "расширенные возможности **shmat**" относятся только к 32-разрядным процессам.

Совместимость mmap со стандартами

Службы mmap описаны в различных стандартах и обычно служат интерфейсом отображения файлов для других реализаций операционной системы. Однако реализации процедуры **mmap** в различных системах могут быть различными. Ниже перечислены отличия данной реализации процедуры **mmap**:

- Отображение в закрытую область данных процесса не поддерживается.
- Обратное неявное отображение не выполняется. Операция **mmap** с указанием **MAP_FIXED** вызывает ошибку, если в указанном диапазоне уже есть отображение.
- При работе с закрытыми образами опция записи по команде копирует измененную страницу на диск при первой же ссылке на команду записи.
- Отображение блоков памяти ввода-вывода или памяти устройств не поддерживается.
- Отображение устройств символьного вывода, а также применение области **mmap** в качестве буфера для операций чтения-записи для таких устройств не поддерживается.
- Функция **madvise** поддерживается только для совместимости. Система не выполняет никаких действий в соответствии с полученной информацией.
- С помощью процедуры **mprotect** можно размещать непреобразованные страницы в указанной области памяти. При обработке непреобразованные страницы просто пропускаются.
- Опции OSF/AES для явного отображения по умолчанию, а также для флагов **MAP_INHERIT**, **MAP_HASSEMAPHORE** и **MAP_UNALIGNED** не поддерживаются.

Процедуры семафоров

Процедуры **msem_init**, **msem_lock**, **msem_unlock** , **msem_remove**, **msleep** и **mwakeup** соответствуют спецификации OSF Application Environment. Они представляют собой альтернативу таким интерфейсам IPC, как **semget** и **semop**. К преимуществам семафоров относятся эффективный метод сериализации и снижение нагрузки благодаря тому, что системный вызов при обращении к семафору не нужен.

Семафоры необходимо размещать в общих областях памяти. Семафоры задаются в структурах **msemaphore**. Все значения в структуре **msemaphore** должны быть результатами вызова процедуры **msem_init**. После вызова этой процедуры могут следовать (но это необязательно) несколько вызовов процедуры **msem_lock** или **msem_unlock**. Если структура **msemaphore** создается другим способом, то результаты вызова процедур семафоров будут не определены.

Адрес структуры **msemaphore** имеет особое значение. Не изменяйте его вручную. Если данная структура содержит значения, скопированные из структуры **msemaphore**, расположенной по другому адресу, то результаты вызова процедур семафоров будут не определены.

Производительность процедур семафоров может снизиться в случае, если структуры семафоров размещены в неименованных областях памяти, созданных процедурой **mmap**. В частности, если много процессов ссылаются на одни и те же семафоры. В таком случае следует разместить структуры семафоров за пределами общих областей памяти, созданных процедурами **shmget** и **shmat**.

Отображение файлов с помощью процедуры shmat

Отображение файлов может применяться для снижения нагрузки, вызванной записью и чтением содержимого файлов. После отображения содержимого файла в пользовательскую область памяти с этим файлом можно работать так же, как с данными в памяти, т.е. с помощью указателей, а не операций ввода-вывода. Копия файла на диске выступает также в роли пространства подкачки для этого файла.

Программа может работать с обычным файлом как с отображенным файлом данных. Можно также применить отображение к файлам, содержащим объектный и исполняемый код. Поскольку на обращение к отображенным файлам затрачивается значительно меньше времени, чем к обычным, отображение исполняемого файла программы значительно ускоряет ее загрузку.

Для создания отображенного исполняемого файла программы скомпилируйте и скомпонуйте программу, вызвав команду **cc** или **ld** с флагом **-K**. Флаг **-K** указывает компоновщику, что следует создать объектный файл с выровненными страницами, т.е. каждый компонент объектного файла должен начинаться с новой страницы (адрес начала фрагмента должен нацело делиться на 2 Кб). В результате объектный файл будет содержать пустые фрагменты, но будет готов к отображению в память. При отображении объектного файла в память обработка текста и данных выполняется по-разному.

Отображенные файлы с записью по команде

Для того чтобы изменения в отображенных файлах не заносились сразу же в их копии на диске, применяется отображение с записью по команде. Изменения в таких файлах сохраняются в системном пространстве подкачки, а не в исходном файле на диске. Для сохранения изменений необходимо явно вызвать команду записи на диск, в противном случае при закрытии файла изменения будут потеряны.

Поскольку в этом случае изменения в файле не заносятся на диск сразу же, отображенные файлы с записью по команде следует применять только в среде взаимодействующих процессов.

Система не распознает конец файла, отображенного с помощью процедуры **shmat**. Поэтому, если программа при занесении данных в отображенный файл с записью по команде выходит за его текущую границу, в исходный файл на диске будет дописан блок нулей соответствующего размера. Если программа не вызывает процедуру **fsync** перед закрытием файла, то данные, записанные в область файла, которая находится за предыдущим концом файла, не будут перенесены на диск. Размер файла увеличится, но в конце файла будут стоять нули. Поэтому всегда вызывайте процедуру **fsync** перед закрытием отображенного файла с записью по команде.

Отображение общих сегментов памяти с помощью процедуры shmat

Способ обработки общих сегментов памяти в системе сходен со способом создания и использования файлов. Для понимания принципов работы с отображенными общими сегментами памяти необходимо разобраться в применяемой терминологии. Ниже приведен список терминов, употребляемых при работе с общими сегментами памяти:

Термин	Определение
ключ	Уникальный идентификатор общего сегмента памяти. Ключ связан с общим сегментом памяти с момента его создания до момента уничтожения. Ключ аналогичен *имени файла*.
ИД сегмента (shmid)	Идентификатор общего сегмента памяти для конкретного процесса. Аналогичен *дескриптору файла*.
Подключить	Указывает, что процесс для работы с сегментом должен подключить его. Подключение общего сегмента аналогично открытию файла.
Отключить	Указывает что процесс должен отключить сегмент по окончании работы с ним. Отключение общего сегмента аналогично закрытию файла.

Понятия, связанные с данным:

“Адресное пространство программы - обзор” на стр. 600
В состав базовой операционной системы включен набор служб распределения памяти между приложениями.

“Создание общего сегмента памяти с помощью функции shmat” на стр. 612
В этом разделе приведены инструкции по созданию общего сегмента памяти с помощью функции **shmat**.

“Ограничения средств межпроцессной связи”
В этом разделе рассмотрены системные ограничения для механизмов взаимодействия между процессами (IPC).

Ограничения средств межпроцессной связи

В этом разделе рассмотрены системные ограничения для механизмов взаимодействия между процессами (IPC).

В некоторых системах UNIX системные администраторы могут изменять ограничения средств IPC (семафоров, сегментов общей памяти и очередей сообщений) с помощью файла **/etc/master**. Недостаток этой возможности заключается в том, что чем выше ограничения средств IPC, тем больше ресурсов выделяется ядру, а это может отрицательно сказаться на производительности системы.

В AIX применяется другой подход. В AIX верхние ограничения для IPC не настраиваются. Структуры данных IPC выделяются и удаляются по необходимости, поэтому требования к памяти зависят от текущего состояния IPC.

Разница в этих подходах зачастую вводит пользователей, работающих с базами данных, в заблуждение. Проблемы могут быть вызваны только одним ограничением - максимальным количеством сегментов общей памяти на процесс. Для 64-разрядных процессов максимальное число сегментов общей памяти равно 268435456. Для 32-разрядных процессов максимальное число сегментов общей памяти равно 11, если только не используется расширенная функция **shmat**.

Ограничения на семафоры для IPC приведены в следующих таблицах.

Семафоры	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	7.1
Макс. число ИД семафоров в 32-разрядном ядре	4096	4096	131072	131072	131072	131072	нет
Макс. число ИД семафоров в 64-разрядном ядре	4096	4096	131072	131072	131072	1048576	1048576
Макс. число семафоров на ИД	65535	65535	65535	65535	65535	65535	65535
Макс. число операций на вызов semop	1024	1024	1024	1024	1024	1024	1024
Макс. число записей отката на процесс	1024	1024	1024	1024	1024	1024	1024
Размер структуры отката в байтах	8208	8208	8208	8208	8208	8208	8208
Макс. значение семафора	32767	32767	32767	32767	32767	32767	32767
Макс. значение коррекции на выходе	16384	16384	16384	16384	16384	16384	16384

Ограничения на очереди сообщений для IPC приведены в следующих таблицах.

Очередь сообщений	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	7.1
Макс. размер сообщения	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ
Макс. число байт в очереди	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ	4 МБ
Максимальное число ИД очередей сообщений в 32-разрядном ядре	4096	4096	131072	131072	131072	131072	131072
Максимальное число ИД очередей сообщений в 64-разрядном ядре	4096	4096	131072	131072	131072	1048576	1048576
Макс. число очередей сообщений на ИД	524288	524288	524288	524288	524288	524288	524288

Ограничения на общую память для IPC приведены в следующих таблицах.

Общая память	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	7.1
Макс. размер сегмента (32-разрядные процессы)	256 МБ	2 ГБ	2 ГБ	2 ГБ	2 ГБ	2 ГБ	2 ГБ
Макс. размер сегмента (64-разрядные процессы) в 32- разрядном ядре	256 МБ	2 ГБ	2 ГБ	64 ГБ	1 ТБ	1 ТБ	нет
Макс. размер сегмента (64-разрядные процессы) в 64- разрядном ядре	256 МБ	2 ГБ	2 ГБ	64 ГБ	1 ТБ	32 ТБ	32 ТБ
Мин. размер сегмента	1	1	1	1	1	1	1

Общая память	4.3.0	4.3.1	4.3.2	5.1	5.2	5.3	7.1
Макс. число ИД общей памяти (32-разрядное ядро)	4096	4096	131072	131072	131072	131072	131072
Макс. число ИД общей памяти (64-разрядное ядро)	4096	4096	131072	131072	131072	1048576	1048576
Макс. число сегментов на процесс (32-разрядные процессы)	11	11	11	11	11	11	11
Макс. число сегментов на процесс (64-разрядные процессы)	268435456	268435456	268435456	268435456	268435456	268435456	268435456

Примечание: Если для 32-разрядных используется расширенная функция **shmat**, то максимальное число сегментов в процессе ограничено только размером адресного пространства

Ограничения IPC в AIX4.3

- Ограничения для семафоров и очередей сообщений показаны в таблице.
- Максимальный размер сегмента общей памяти равен 256GB.
- Для общей памяти без расширенной функции **shmat**:
  - Максимальное число сегментов в процессе равно 11.
- Для общей памяти с расширенной функцией **shmat**:
  - При подключении сегмента общей памяти его размер округляется до значения, кратного 4096 байт.
  - Процесс может подключать сегменты общей памяти до тех пор, пока они помещаются в адресном пространстве. Максимальный размер адресного пространства равен 11 сегментам, то есть 11, помноженному на 256 МБ.
- Расширенная функция **shmat** применяется в случае, если переменной среды **EXTSHM** присвоено значение ON при запуске процесса.
- При использовании модели большого или очень большого адресного пространства объем доступного адресного пространства для общих сегментов сокращается.

Ограничения IPC в AIX 4.3.1

- Максимальный размер сегмента общей памяти увеличен с 256 МБ до 2 ГБ. При подключении сегмента общей памяти размером больше 256 МБ, его размер округляется до значения, кратного 256 МБ, даже если применяется расширенная функция **shmat**.

Ограничения IPC в AIX 4.3.2

- Максимальное число очередей сообщений, ИД семафоров и сегментов общей памяти равно 131072.
- Максимальное число сообщений в очереди равно 524288.

Ограничения IPC в AIX 5.1

- Максимальный размер сегмента общей памяти для 64-разрядных процессов равен 64 ГБ. 32-разрядные процессы не могут подключать сегменты общей памяти, большие 2 ГБ.

Ограничения IPC в AIX 5.2

- Максимальный размер сегмента общей памяти для 64-разрядных процессов равен 1 ТБ. 32-разрядные процессы не могут подключать сегменты общей памяти, большие 2 ГБ.
- С помощью **shmat** 32-разрядные приложения могут получить более 11 сегментов общей памяти при работе с моделью очень большого адресного пространства без расширения **shmat**. Дополнительная информация о модели очень большого адресного пространства.
- Приложения могут узнать ограничения для IPC с помощью системного вызова **vmgetinfo**.

Ограничения IPC в AIX 5.3

- Максимальный размер сегмента общей памяти для 64-разрядных процессов равен 32 ТБ. 32-разрядные процессы не могут подключать сегменты общей памяти, большие 2 ГБ.
- С помощью **shmat** 32-разрядные приложения могут получить более 11 сегментов общей памяти при работе с моделью очень большого адресного пространства без расширения **shmat**. Дополнительная информация о модели очень большого адресного пространства.
- Приложения могут узнать ограничения для IPC с помощью системного вызова **vmgetinfo**.

Ограничения IPC в AIX 6.1

32-разрядное ядро в AIX 6.1 уже не поддерживается. Все остальные значения остаются без изменений.

Понятия, связанные с данным:

“Отображение памяти - основные сведения” на стр. 602
Скорость обработки команд приложения в системе обратно пропорциональна числу операций считывания данных, хранящихся за пределами адресуемой программами памяти.

Создание отображенных файлов данных с помощью функции shmat

В этом разделе приведены инструкции по созданию отображенного файла с помощью функции **shmat**.

Предварительные требования

Отображаемый файл должен быть обычным файлом.

Процедура

Создание отображенного файла - процесс, состоящий из двух этапов. Сначала нужно создать отображенный файл. Затем необходимо разработать метод определения конца файла, так как процедура **shmat** не поддерживает эту функцию.

1. Для создания отображенного файла:
   a. Откройте (или создайте) файл и сохраните его дескриптор:
   ```
   if( ( fildes = open( filename , 2 ) ) < 0 )
   {
           printf( "не удалось открыть файл\n" );
           exit(1);
   }
   ```
   b. Отобразите файл в сегмент с помощью функции **shmat**:
   ```
   file_ptr=shmat (fildes, 0, SHM_MAP);
   ```
   Константа `SHM_MAP` определена в файле **/usr/include/sys/shm.h**. Эта константа указывает, что файл - отображенный. С помощью следующей директивы подключите этот файл и другие файлы заголовков для работы с общей памятью:
   ```
   #include <sys/shm.h>
   ```
2. Для обнаружения конца отображенного файла:
   a. С помощью функции **lseek** перейдите в конец файла:

```
eof = file_ptr + lseek(fildes, 0, 2);
```

В этом примере значение `eof` устанавливается равным адресу конца файла + 1. Применяйте это значение в качестве маркера конца файла.

b. Считайте, что `file_ptr` указывает на начало файла данных, и работайте с данными так, как будто они расположены в памяти:

```
while ( file_ptr < eof)
{
      .
      .
      .
      (работа с файлом через указатель file_ptr)
}
```

Примечание:Процедуры **read** и **write** могут работать с отображенными файлами, при этом будут получены те же результаты, что и при работе через указатели.

c. Закройте файл, когда программа завершит работу с ним:

```
close (fildes );
```

Создание отображенного файла с записью по команде с помощью функции shmat

В этом разделе приведены инструкции по созданию отображенного файла с записью по команде с помощью функции **shmat**.

Предварительные требования

Отображаемый файл должен быть обычным файлом.

Процедура

1. Откройте (или создайте) файл и сохраните его дескриптор:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
        printf( "не удалось открыть файл\n" );
        exit(1);
}
```

2. Отобразите файл в сегмент в режиме записи по команде с помощью функции **shmat**:

```
file_ptr = shmat( fildes, 0, SHM_COPY );
```

Константа `SHM_COPY` определена в файле **/usr/include/sys/shm.h**. Эта константа указывает, что файл - отображенный с записью по команде. С помощью следующей директивы подключите этот файл и другие файлы заголовков для работы с общей памятью:

```
#include <sys/shm.h>
```

3. Считайте, что `file_ptr` указывает на начало файла данных, и работайте с данными так, как будто они расположены в памяти.

```
while ( file_ptr < eof)
{
      .
      .
      .
      (работа с файлом через указатель file_ptr)
}
```

4. Для записи внесенных в файл изменений на диск применяйте функцию **fsync**:

```
fsync( fildes );
```

5. Закройте файл, когда программа завершит работу с ним:

```
close (fildes );
```

Создание общего сегмента памяти с помощью функции shmat

В этом разделе приведены инструкции по созданию общего сегмента памяти с помощью функции **shmat**.

Предварительные требования

Нет.

Процедура

1. Создайте ключ для идентификации общего сегмента. Для этого вызовите функцию **ftok**. Например, для создания ключа `mykey` с идентификатором проекта `R`, хранящимся в переменной `proj` (тип **char**) и именем файла `null_file`, воспользуйтесь следующим оператором:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Выполните одно из следующих действий:
   - Создайте общий сегмент памяти с помощью функции **shmget**. Например, для создания сегмента размером 4096 и присвоения значения **shmid** целочисленной переменной `mem_id`, используйте следующее выражение:

     ```
     mem_id = shmget(mykey, 4096, IPC_CREAT | o666 );
     ```

   - Получите указатель на ранее созданный общий сегмент с помощью функции **shmget**. Например, для того чтобы получить ссылку на сегмент, ранее связанный с ключом `mykey` и присвоить значение **shmid** целочисленной переменной `mem_id`, укажите в программе оператор:

     ```
     mem_id = shmget( mykey, 4096, IPC_ACCESS );
     ```

3. Предоставьте общий сегмент процессу с помощью функции **shmat**. Например, для подключения созданного сегмента можно воспользоваться оператором

   ```
   ptr = shmat( mem_id );
   ```

   В этом примере переменная `ptr` представляет собой указатель на структуру, определяющую поля общего сегмента. Такая структура-шаблон применяется для чтения и записи данных в общий сегмент. Шаблон должен совпадать для всех процессов, работающих с общим сегментом.
4. Выполните необходимые операции с данными сегмента, пользуясь шаблонной структурой.
5. Отключите общий сегмент от процесса с помощью функции **shmdt**:

   ```
   shmdt( ptr );
   ```

6. Если общий сегмент больше не нужен, удалите его с помощью функции **shmctl**:

   ```
   shmctl( mem_id, IPC_RMID, ptr );
   ```

   > **Примечание:** с помощью команды **ipcs** можно получить информацию о сегменте, а с помощью команды **ipcrm** удалить сегмент.

Понятия, связанные с данным:

“Отображение памяти - основные сведения” на стр. 602
Скорость обработки команд приложения в системе обратно пропорциональна числу операций считывания данных, хранящихся за пределами адресуемой программами памяти.

Требования программ к пространству подкачки

Объем пространства подкачки, который требуется приложению, зависит от типа действий, выполняемых в системе. При нехватке пространства подкачки возможно аварийное завершение работы процессов.

Если пространство подкачки переполнится, может произойти сбой в работе системы. При уменьшении свободной области в пространстве подкачки необходимо определить дополнительную область подкачки.

Система отслеживает число свободных блоков пространства подкачки и предупреждает об их нехватке. Команда **vmstat** позволяет получить статистическую информацию, связанную с работой виртуальной

памяти. Когда число свободных блоков становится меньше установленного порога, называемого уровнем предупреждения пространства подкачки, система информирует об этом все процессы (кроме **kprocs**), отправляя сигнал **SIGDANGER**.

Примечание: Если объем свободного пространства подкачки оказывается меньше второго установленного порога, называемого критическим уровнем пространства подкачки, система отправляет сигнал **SIGKILL** процессам, использующим большую часть пространства подкачки и не обрабатывающим сигнал **SIGDANGER**. По умолчанию сигнал **SIGDANGER** игнорируется. Система завершает процессы с помощью сигнала **SIGKILL** до тех пор, пока число свободных блоков в пространстве подкачки не выйдет из критического диапазона. Если параметру **low_ps_handling** присвоено значение 2 (с помощью команды **vmo**) и при этом нет процессов, подлежащих уничтожению (без обработчика **SIGDANGER**), то система отправит сигнал **SIGKILL** первым из процессов, у которых есть обработчик сигнала **SIGDANGER**.

Процессы, динамически запрашивающие память, могут проверять наличие достаточного объема пространства подкачки, отслеживая значения уровней пространства подкачки с помощью функции **psdanger** или специальных функций выделения памяти. Процессы могут избежать принудительного завершения, определив обработчик сигнала **SIGDANGER** и выполнив функцию **disclaim** для освобождения памяти и ресурсов пространства подкачки, выделенных в областях данных и стека, а также в сегментах общей памяти.

Ниже перечислены другие функции, позволяющие получить информацию о пространстве подкачки у VMM:

Функция	Описание
mincore	Позволяет получить информацию о выгруженных страницах памяти.
madvise	Позволяет процессу сообщить системе о предполагаемом использовании пространства подкачки.
swapqry	Возвращает состояние устройства подкачки.
swapon	Запускает процесс обмена данными с указанным блочным устройством.

Понятия, связанные с данным:

“Адресное пространство программы - обзор” на стр. 600
В состав базовой операционной системы включен набор служб распределения памяти между приложениями.

Список функций для отображения памяти

Данные функции работают с массивами символов, называемыми областями памяти.

Они позволяют выполнить следующие действия:

- Поместить символ в область памяти
- Скопировать символы из одной области памяти в другую
- Сравнить содержимое областей памяти
- Установить размер области памяти

При компилировании программы, применяющей функции работы с памятью, специальных флагов указывать не нужно. Однако программа должна содержать заголовочный файл с определениями этих функций. Для включения заголовочного файла добавьте следующую строку:

```
#include <memory.h>
```

Ниже приведено описание функций для работы с памятью:

Служба	Описание
compare_and_swap	Сравнивает данные и записывает их на диск
fetch_and_add	Обновляет переменную длиной в слово; выполняется как атомарная операция
fetch_and_and или **fetch_and_or**	Устанавливает или сбрасывает биты переменной длиной в слово; выполняется как атомарная операция
malloc, **free**, **realloc**, **calloc**, **mallopt**, **mallinfo**, и **alloca**	Выделяет память
memccpy, **memchr**, **memcmp**, **memcpy**, **memset или memmove**	Операции с памятью.

Служба	Описание
moncontrol	Запускает профайлер после инициализации с помощью функции **monitor** или завершает его работу
monitor	Запускает профайлер с помощью данных, заданных в параметрах функции, или завершает его работу
monstartup	Запускает профайлер с областями данных, размер которых устанавливается по умолчанию, либо завершает его работу
mprotect	Изменяет права доступа указанного диапазона адресов сегмента общей памяти.
msem_init	Инициализирует семафор в отображенном файле или области общей памяти
msem_lock	Блокирует семафор
msem_remove	Удаляет семафор
msem_unlock	Разблокирует семафор
msleep	Переводит процесс в режим ожидания, если семафор занят
mwakeup	Активизирует процесс, ожидающий семафор
disclaim	Аннулирует содержимое ячеек памяти с адресами из заданного диапазона
ftok	Генерирует стандартный ключ для взаимодействия процессов
getpagesize	Возвращает системный размер страницы
psdanger	Задает объем свободного пространства подкачки
shmat	Выделяет текущему процессу сегмент общей памяти или отображенный файл
shmctl	Управляет операциями с общими областями памяти
shmdt	Освобождает сегмент общей памяти
shmget	Возвращает сегмент общей памяти
swapon	Активизирует обмен данными с указанным блочным устройством
swapqry	Возвращает информацию о состоянии устройства

Список функций для отображения памяти

Функции отображения памяти работают с областями памяти, отображенными функцией **mmap**.

Они позволяют выполнить следующие действия:

- Отобразить объектный файл в виртуальную память
- Синхронизировать отображенный файл
- Определить расположение страниц памяти
- Определить права доступа к отображенной области памяти
- Освободить отображенные области памяти.

При компиляции программы, применяющей функции для работы с памятью, специальных флагов указывать не нужно. Однако для работы с некоторыми из этих функций необходимо включить в программу файл заголовка. Если в описании функции указан файл заголовка, то его необходимо указать в программе следующим образом:

```
#include <HeaderFile.h>
```

Ниже перечислены функции отображения памяти:

Служба	Описание
madvise	Сообщает системе о предположительном объеме подкачки для процесса.
mincore	Определяет расположение страниц памяти.
mmap	Отображает объектный файл в виртуальную память.
mprotect	Изменяет права доступа к отображенной области памяти.
msync	Синхронизирует копию отображенного файла с самим файлом, хранящемся на запоминающем устройстве.
munmap	Освобождает область памяти, выделенную для отображения.

Векторное программирование AIX

В некоторых процессорах PowerPC реализована поддержка векторного расширения на основе операций Single Instruction Multiple Data (SIMD).

Векторное расширение архитектуры PowerPC (зачастую называется AltiVec или VMX) предоставляет дополнительный набор инструкций для выполнения векторно-матричных математических функций.

Логический модуль векторной арифметики представляет собой стандартный модуль арифметики SIMD, в котором отдельная инструкция предусматривает выполнение одной операции над всеми элементами данных каждого вектора. AIX 5.3 с рекомендуемым уровнем обслуживания 5300-30 представляет собой первый выпуск AIX, поддерживающий векторное программирование. Процессор IBM PowerPC 970 - это первый поддерживаемый в AIX процессор, в котором реализовано векторное расширение. В настоящее время такими процессорами комплектуются одноплатные серверы JS20, предлагаемые в рамках решений BladeCenter.

Обзор векторного расширения

В состав векторного расширения входит дополнительный набор из 32 128-разрядных регистров, предназначенных для записи различных векторов. Например, поддерживаются 8-, 16- и 32-разрядные целые числа со знаком и без, а также 32-разрядные числа с плавающей точкой IEEE одинарной точности. Кроме того, предусмотрен регистр управления и состояния, содержащий бит привязки состояния, указывающий на насыщенность, а также бит управления, позволяющий выбрать режим Java для операций над числами с плавающей точкой.

По умолчанию для каждого процесса AIX инициализирует режим Java, в котором все операции над числами с плавающей точкой соответствуют стандарту IEEE. Альтернативный режим обеспечивает меньшую точность вычислений над числами с плавающей точкой, однако в некоторых случаях он позволяет значительно повысить скорость вычислений. Например, процессор PowerPC 970, работающий в режиме Java, при обработке некоторых инструкций над векторами с плавающей точкой может выдать исключительную ситуацию, если входные операнды или результат являются ненормированными значениями, для обработки которых необходимы дополнительные ресурсы операционной системы. По этой причине рекомендуется явным образом устанавливать альтернативный режим, если округление допустимо, либо избегать векторных вычислений над ненормированными значениями.

Кроме того, векторное расширение включает в себя более 160 инструкций, обеспечивающих доступ к векторным регистрам и памяти в операциях управления регистрами, операциях вещественной и целочисленной арифметики, логических операциях и операциях сравнения векторов. В операциях вещественной арифметики применяется формат IEEE 754-1985 с одинарной точностью без поддержки исключительных ситуаций IEEE. Для всех исключительных ситуаций выдаются результаты по умолчанию, указанные в IEEE для необрабатываемых исключений. Поддерживается только режим IEEE по умолчанию (округление до ближайшего). Вместо инструкций деления и извлечения квадратного корня для чисел с плавающей точкой предусмотрены соответствующие инструкции оценки.

Кроме того, предусмотрен 32-разрядный регистр специального назначения, управляемый программным обеспечением, в котором представлена битовая маска используемых векторных регистров. Такой подход позволяет операционной системе оптимизировать алгоритмы сохранения и восстановления векторов в ходе управления переключением контекста.

Динамическая проверка поддержки операций над векторами

Программа может определить наличие в системе поддержки векторного расширения в соответствии со значением поля **vmx_version**, входящего в состав структуры _system_configuration. Если в этом поле указано значение, отличное от нуля, то процессоры и операционная система поддерживают векторное расширение. Для выполнения этой проверки предусмотрен макрос __power_vmx(), расположенный в файле **/usr/include/sys/systemcfg.h**. Такая возможность удобна для программного обеспечения, которое использует векторное расширение при его наличии, либо эквивалентные скалярные методы в случае его отсутствия.

Расширение ABI AIX

Двоичный интерфейс приложений (ABI) AIX расширен для поддержки дополнительных состояний и соглашений векторных регистров. Полное описание расширений ABI приведено в *Справочник по ассемблеру*.

AIX поддерживает спецификацию программного интерфейса AltiVec. Ниже приведена таблица векторных типов данных C/C++. Размер всех векторных типов данных составляет 16 байт и предусматривает выравнивание по границе 16-байтового слова. Объекты, содержащие векторные типы, должны подчиняться стандартным соглашениям по выравниваю в соответствии с требованием наибольшего элемента. Если объект, содержащий векторный тип, упакован, то гарантия 16-байтового выравнивания отсутствует. Требуется компилятор AIX, поддерживающий спецификацию программного интерфейса AltiVec.

Таблица 78. Новые векторные типы данных C/C++

Новые типы C/C++	**Содержимое**
vector unsigned char	16 символов без знака
vector signed char	16 символов со знаком
vector bool char	16 символов без знака
vector unsigned short	8 коротких чисел без знака
vector signed short	8 коротких чисел со знаком
vector bool short	8 коротких чисел без знака
vector unsigned int	4 целых без знака
vector signed int	4 целых со знаком
vector bool int	4 целых без знака
vector float	4 числа с плавающей точкой

В следующей таблице рассмотрены соглашения по применению векторных регистров.

Таблица 79. Соглашения по применению векторных регистров

Тип регистра	**Регистр**	**Состояние**	**Использование**
VRs	VR0	Динамический	Рабочий регистр
	VR1	Динамический	Рабочий регистр
	VR2	Динамический	Первый аргумент вектора Первый вектор значения, возвращаемого функцией
	VR3	Динамический	Второй аргумент вектора, рабочий регистр
	VR4	Динамический	Третий аргумент вектора, рабочий регистр

Таблица 79. Соглашения по применению векторных регистров (продолжение)

Тип регистра	Регистр	Состояние	Использование
	VR5	Динамический	Четвертый аргумент вектора, рабочий регистр
	VR6	Динамический	Пятый аргумент вектора, рабочий регистр
	VR7	Динамический	Шестой аргумент вектора, рабочий регистр
	VR8	Динамический	Седьмой аргумент вектора, рабочий регистр
	VR9	Динамический	Восьмой аргумент вектора, рабочий регистр
	VR10	Динамический	Девятый аргумент вектора, рабочий регистр
	VR11	Динамический	Десятый аргумент вектора, рабочий регистр
	VR12	Динамический	Одиннадцатый аргумент вектора, рабочий регистр
	VR13	Динамический	Двенадцатый аргумент вектора, рабочий регистр
	VR14:19	Динамический	Рабочий регистр
	VR20:31	Зарезервирован (режим по умолчанию) Статический (расширенный режим ABI)	В режиме с поддержкой векторов по умолчанию эти регистры зарезервированы и недоступны для применения. В расширенном режиме ABI с поддержкой векторов - это статические регистры, значения которых сохраняются при вызове функций.
Специальное назначение	VRSAVE	Зарезервировано	VRSAVE не применяется в ABI AIX. Совместимая с ABI программа не должна использовать либо изменять VRSAVE.
Специальное назначение	VSCR	Динамический	Регистр управления и состояния, содержащий бит состояния насыщенности, а также бит управления режимом Java.

В спецификации программного интерфейса AltiVec определяется регистр VRSAVE, который следует применять в качестве битовой маски для регистров векторов. В AIXзапрещено изменение регистра VRSAVE приложением.

Первые 12 параметров векторов, обрабатываемых функцией, размещаются в регистрах VR2 - VR13. Неиспользуемые регистры параметров векторов в момент входа в функцию содержат неопределенные значения. Параметры векторов из списка аргументов фиксированной длины в регистры общего назначения (GPR) не копируются. Дополнительные параметры векторов, начиная с тринадцатого, передаются с помощью памяти стека программы в подходящем преобразованном расположении области параметров в соответствии с их положением в списке параметров. Кроме того, к ним применяется 16-байтовое выравнивание.

В случае списков аргументов переменной длины **va_list** продолжает служить указателем на расположение в памяти, содержащее следующий параметр. При обращении к векторному типу с помощью **va_arg()** следует

предварительно выровнять **va_list** по границе 16-байтового слова. Получатель списка аргументов переменной длины отвечает за выполнение выравнивания перед извлечением параметра векторного типа.

Неупакованные структуры и объединения, передаваемые значением, в состав которого входит векторный элемент, выравниваются по границе 16-байтового слова в стеке.

Все параметры функции, принимающей список аргументов переменной длины, преобразуются в области аргументов, упорядочиваются и выравниваются в соответствии с их типами. Первые восемь слов (32-бита) или двойных слов (64-бита) списка аргументов переменной длины размещаются в регистрах общего назначения r3 - r10. Сюда же включены параметры векторов.

Функции возвращают значения векторного типа данных с помощью регистра VR2. Все функции, возвращающие данные векторных типов, либо применяющие параметры векторов, должны иметь прототипы. В общем случае это позволяет избежать копирования векторных регистров в регистры общего назначения.

Совместимость и стыкуемость со старыми версиями ABI

Принцип работы интерфейсов, предусматривающих сохранение и восстановление статического состояния системы (например, setjmp(), longjmp(), sigsetjmp(), siglongjmp(), _setjmp(), _longjmp(), getcontext(), setcontext(), makecontext(), и swapcontext()), обуславливает некоторый риск при рассмотрении зависимостей между старым и расширенным модулями ABI. Ситуация осложняется еще и тем, что семейство функций setjmp входит в состав статического элемента библиотеки libc. Таким образом, каждый двоичный объект AIX имеет статическую копию семейства setjmp и прочих функций связанной версии AIX. Более того, определения структур данных jmpbufs и ucontext, содержащиеся в существующих двоичных объектах AIX, не позволяют разместить дополнительные состояния статических векторных регистров.

Во всех ситуациях, когда старые и новые модули чередуют вызовы или обратные вызовы, когда старый модуль может выполнить функцию longjmp() или setcontext() в обход стандартного соглашения компоновки расширенных модулей, существует вероятность потери состояния статического векторного регистра.

По этой причине, если ABI AIX определяет статические векторные регистры, то при работе с векторами (AltiVec) в компиляторах AIX по умолчанию применяется режим, запрещающий применение статических векторных регистров. Такой подход по умолчанию обеспечивает среду компиляции, позволяющую работать с векторами (AltiVec) без риска, связанного со стыкуемостью с устаревшими модулями.

Для приложений, особенности стыкуемости и зависимости модулей которых полностью известны, можно указать дополнительную опцию компиляции, разрешающую использование статических векторных регистров. Этот режим следует применять только в том случае, если известны все зависимые старые модули, а также особенности их работы. В частности, не должны применяться такие функции, как setjmp(), sigsetjmp(), _setjmp() и getcontext(), все переходы должны выполняться в соответствии со стандартным соглашением компоновки функций и должны отсутствовать обратные вызовы к предыдущей версии модуля.

Среда компиляции AltiVec по умолчанию определяет **__VEC__** в соответствии с описанием, приведенным в руководстве *AltiVec Technology Programming Interface Manual*.

Если применяется опция, разрешающая применение статических векторных регистров, то среда компиляции должна дополнительно определять **__EXTABI__**. При компиляции модулей, в которых векторы не применяются, поддержку ABI можно указать явным образом, определив **__AIXEXTABI**. В этом случае такие модули смогут безопасно взаимодействовать с модулями с поддержкой векторов, в которых разрешено применение статических регистров.

Расширенный контекст

Для поддержки дополнительного состояния системы, необходимого для векторного расширения, а также для других расширений, таких как пользовательские клавиши, в AIX 5.3 предусмотрена поддержка структур

расширенного контекста. Основное назначение доступного для приложений контекста системы заключается в его присутствии в структуре sigcontext, предоставляемой обработчикам сигналов, с последующей его активацией после возврата sigcontext от обработчика сигналов. Фактически, структура sigcontext представляет собой подмножество структуры ucontext. Различие между двумя структурами заключается только в результате sizeof(struct sigcontext). При создании контекста сигналов, подлежащего передаче обработчику сигналов, AIX фактически создает структуру ucontext на основе стека обработчика сигналов. Контекст системы, входящий в состав контекста сигналов, должен содержать все сведения о динамическом или статическом состоянии системы для случайно прерванного контекста. Для реализации этого без утраты двоичной совместимости с существующими обработчиками сигналов область, ранее зарезервированная в структуре ucontext, служит в качестве индикатора доступности информации расширенного контекста.

Новое поле **__extctx**, определенное в **ucontext**, задает адрес структуры расширенного контекста **struct __extctx** (см. файл **sys/context.h**). Новое поле **__extctx_magic**, входящее в состав структуры ucontext, указывает на допустимость информации расширенного контекста, если для **__extctx_magic** указано значение **__EXTCTX_MAGIC**. Дополнительное состояние системы обработки векторных данных для нити, применяющей векторное расширение, сохраняется и восстанавливается в структуре ucontext вместе с новым расширением контекста в процессе обработки сигналов.

Кроме того, структура ucontext применяется в таких API, как getcontext(), setcontext(), swapcontext() и makecontext(). В этих случаях контекст сохраняется только в ответ на целенаправленное действие и соглашение компоновки требует сохранения только статического состояния системы. Поскольку в режиме AIX с поддержкой векторов по умолчанию запрещено использование статических регистров (см. раздел, посвященный ABI), для большинства приложений расширения структуры ucontext не требуются. Если приложение явным образом разрешает использование статических регистров, оно выбирает расширенную структуру ucontext, в которой предусмотрена память для поля **__extctx**, входящего в неявное определение **__EXTABI__**. Расширенный контекст ucontext можно указать, явным образом определив **__AIXEXTABI**.

Аналогичным образом в режиме с поддержкой векторов по умолчанию jmp_buf можно использовать совместно с setjmp() или longjmp() без дополнительных изменений, поскольку статические векторные регистры не применяются. В результате явного включения статических векторных регистров jmp_buf выделяются большие объемы памяти, обусловленные неявным определением **__EXTABI__** компилятором. Кроме того, расширенные буферы перехода можно активировать путем явного определения __AIXEXTABI.

Структуру информации расширенного контекста можно просмотреть в заголовочном файле **sys/context.h**.

Выделение и выравнивание памяти для векторных данных

Векторные типы данных требуют выравнивания по границе 16-байтового слова. В соответствии со спецификацией программного интерфейса AltiVec для выделения памяти с 16-байтовым выравниванием в AIX предусмотрен набор функций malloc (vec_malloc, vec_free, vec_realloc, vec_calloc).

В ходе компиляции в режиме с поддержкой векторов, если **_VEC_** неявно определен компилятором, вызовы устаревших функций malloc и calloc перенаправляются к эквивалентным функциям vec_malloc и vec_calloc с поддержкой векторов. Исходный код, в котором векторы не применяются, также можно скомпилировать с добавлением перенаправлений malloc и calloc, определив **__AIXVEC** явным образом. Выравнивание памяти, выделяемой malloc(), realloc() и calloc() по умолчанию, также можно контролировать в динамическом режиме.

Во-первых за пределами любой программы можно задать переменную MALLOCALIGN, указав в ней выравнивание по умолчанию, необходимое для каждой операции выделения памяти malloc(). Пример:

```
MALLOCALIGN=16; export MALLOCALIGN
```

В переменной среды MALLOCALIGN можно указать любое значение, кратное двум и превышающее размер указателя в соответствующем режиме выполнения или равное ему (4 байта для 32-разрядного режима и 8

байт для 64-разрядного режима). Если для переменной MALLOCALIGN указано недопустимое значение, оно округляется в большую сторону до ближайшего числа кратного двум и применяется для выравнивания во всех последующих операциях malloc().

Кроме того, внутри программы с помощью новой опции команды mallopt() можно указать предпочитаемое выравнивание для последующих операций выделения памяти. Пример:

```
rc = mallopt(M_MALIGN, 16);
```

Дополнительная информация приведена в описании mallopt и MALLOCALIGN.

Векторные типы данных в функциях printf и scanf

В соответствии со спецификацией программного интерфейса AltiVec поддержка строк формата преобразования векторов добавлена в следующие функции AIX: scanf, fscanf, sscanf, wsscanf, printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf и vwsprintf. Ниже перечислены новые функции форматирования размера:

- vl или lv - использует один аргумент и изменяет существующее целочисленное преобразование, возвращая тип vector signed int, vector unsigned int или vector bool в случае входных преобразований и vector signed int * или vector unsigned int * в случае выходных преобразований. Данные обрабатываются как наборы четырех 4-байтовых компонентов, к каждому из которых применяется соответствующий формат преобразования.
- vh или hv - использует один аргумент и изменяет существующее преобразование коротких целых чисел, возвращая тип vector signed short или vector unsigned short в случае входных преобразований и vector signed short * или vector unsigned short * в случае выходных преобразований. Данные обрабатываются как наборы из восьми 2-байтовых компонентов, к каждому из которых применяется соответствующий формат преобразования.
- v использует один аргумент и изменяет преобразование 1-байтового целого, 1-байтового символа и 4-байтового числа с плавающей точкой. Преобразование числа с плавающей точкой возвращает тип vector float в случае входного преобразования и vector float * для выходного преобразования. Данные обрабатываются как наборы из четырех компонентов 4-байтовых чисел с плавающей точкой, к каждому из которых применяется соответствующий формат преобразования. Преобразование целого числа или символа возвращает тип vector signed char, vector unsigned char или vector bool char в случае входного преобразования или vector signed char * или vector unsigned char * в случае выходного преобразования. Данные обрабатываются как шестнадцать наборов 1-байтовых компонентов, к каждому из которых применяется соответствующий формат преобразования.

Для векторной формы допустим любой формат преобразования, применимый к сингулярной форме векторного типа данных. Преобразования целых чисел %d, %x, %X, %u, %i и %o можно применить вместе со спецификаторами длины вектора %lv, %vl, %hv, %vh и %v. Преобразование символов %c можно применить вместе со спецификатором длины вектора %v. Преобразования чисел с плавающей точкой %a, %A, %e, %E, %f, %F, %g и %G можно применить вместе со спецификатором длины вектора %v.

Для входных преобразований можно указать разделитель, исключающий предшествующие ему пробелы. Если разделитель не указан, по умолчанию в качестве разделителя используется пробел (в том числе пробелы, предшествующие разделителю). Для преобразования символов в качестве разделителя по умолчанию применяется нулевой символ.

Для выходных преобразований разделитель можно указать непосредственно перед преобразованием размера вектора. Если разделитель не указан, то по умолчанию в качестве разделителя применяется пробел. Для преобразования символов в качестве разделителя по умолчанию применяется нулевой символ.

Приложения с поддержкой нитей

Векторное расширение могут использовать приложения с поддержкой нитей. Такие приложения поддерживаются как на уровне системы (модель организации нитей 1:1), так и на уровне процесса (модель организации нитей M:N). Если при компиляции приложения с поддержкой нитей была указана опция,

разрешающая применение статических регистров, то нити pthread этого приложения указываются в качестве расширенных нитей ABI pthread. В результате для таких нитей в библиотеке pthread выделяется больший объем памяти для сохранения контекста. Кроме того, отладчик dbx AIX предоставляет полную поддержку отладки программ, в которых применяются нити и векторы на уровне системы.

Компиляторы

Компилятор AIX, поддерживающий векторное расширение, должен соответствовать спецификации AIX ABI Vector Extension. Как было отмечено ранее, в AIX в режиме компиляции с поддержкой векторов по умолчанию статические векторные регистры выключены. При необходимости вы можете явным образом указать опцию, разрешающую применение статических векторных регистров, предварительно подробно изучив особенности стыкуемости нового и старого модулей, а также связанные риски.

Для разрешения применения статических векторных регистров компилятор C/C++ должен определить **__EXTABI__**. Кроме того, для работы с векторами любого типа компилятор C/C++ должен определить **__VEC__**. В случае компиляции модулей C/C++ без поддержки векторов, которые планируется применять совместно с модулями Fortran с поддержкой векторов, рекомендуется при компиляции модулей C/C++ явным образом определить **__AIXVEC** (явное определение, аналогичное **__VEC__**), а также **__AIXEXTABI** (явное определение, аналогичное **__EXTABI**), если в модулях Fortran применяются статические векторные регистры.

Помимо спецификации программного интерфейса AltiVec, который предоставляет явное расширение языков C/C++ для векторного программирования, некоторые компиляторы допускают применение векторного расширения в параметрах оптимизации процессоров, поддерживающих векторное расширение.

Дополнительная информация приведена в документации по компилятору.

Ассемблер

Компилятор ассемблера AIX, расположенный в каталоге **/usr/ccs/bin/as**, поддерживает новый набор инструкций процессора PowerPC 970, определенный в векторном расширении. Вы можете использовать новый режим сборки -m970 или псевдоопцию .machine 970 в исходном файле для разрешения сборки новых векторных инструкций. Дополнительная информация приведена в *Справочник по ассемблеру*.

Отладчик

Отладчик dbx AIX, расположенный в каталоге /usr/ccs/bin/dbx, поддерживает отладку программ, в которых применяются векторы, на уровне системы. Такая поддержка включает в себя возможность дизассемблирования новых векторных инструкций, а также просмотра и задания векторных регистров. За дизассемблирование инструкций PowerPC 970 (в том числе векторных инструкций, если dbx применяется не в системеPowerPC 970) отвечает новое значение $instructionset 970. Обратите внимание на то, что если dbx выполняется в PowerPC 970, то значение 970 является значением $instructionset по умолчанию.

Поскольку векторные регистры по умолчанию не отображаются, для их просмотра применяется команда unset $novregs. Если процессор, нить или процесс не поддерживают векторное расширение, то состояние векторных регистров не отображается. В противном случае выдается содержимое всех векторных регистров в шестнадцатеричном формате.

Кроме того, можно просмотреть содержимое отдельных векторов в базовом формате. Например, команда print $vr0 позволяет просмотреть содержимое регистра VR0 в виде массива из четырех целых чисел. Команда print $vr0c позволяет просмотреть содержимое регистра VR0 в виде массива из шестнадцати символов. Команда print $vr0s позволяет просмотреть содержимое регистра VR0 в виде массива из восьми коротких чисел, а команда $vr0f - в виде массива из четырех чисел с плавающей точкой.

Вы можете обращаться как к векторным регистрам в целом (например, $vr0 = $vr1), так и к их отдельным элементам аналогично работе с массивами. Например, assign $vr0[3] = 0x11223344 позволяет задать только четвертый целый элемент VR0. assign $vr0f[0] = 1.123 позволяет указать для первого вещественного элемента VR0 значение 1.123.

В ходе выполнения функции или программы векторные регистры можно отслеживать. Например, команда tracei $vr0, указанная для функции main(), отображает содержимое регистра VR0 каждый раз при его изменении в функции main(). Аналогичным образом, указав в команде tracei формат регистра ($vr0f, $vr0c, $vr0s), его содержимое будет отображаться соответствующим образом.

dbx может отображать векторные типы данных в виде массива только в том случае, если компиляторы представляют их в качестве массивов базовых типов данных.

Дополнительная информация приведена в документации по команде **dbx**.

Кроме того, предоставлена поддержка отладчиков других фирм в виде новых операций ptrace PTT_READ_VEC и PTT_WRITE_VEC, позволяющих считывать и записывать состояние векторного регистра на уровне нити. Дополнительная информация приведена в документации по команде ptrace.

Файловая система **/proc** расширена для поддержки отладчика **/proc**. Файлы status и lwpstatus (для процессов и нитей с поддержкой векторов) расширены состоянием векторных регистров. Новое управляющее сообщение PCSVREG позволяет устанавливать состояние векторного регистра путем записи данных в управляющий файл процесса или нити. Дополнительная информация приведена в описании файла **/proc**.

Файлы дампа

AIX поддерживает добавление состояния системы обработки векторных данных в файл дампа процесса или нити с поддержкой векторов. Это состояние добавляется только в том случае, если в соответствующем процессе или нити применяется векторное расширение. Обратите внимание на то, что при выборе форматов версий, предшествующих AIX 4.3, добавление состояний векторов не поддерживается. Поддержка состояния векторов предусмотрена только в текущих форматах файлов дампа. Для чтения файлов дампов и отображения состояния системы обработки векторных данных применяется команда dbx.

Выделение памяти в системе с помощью подсистемы malloc

Для приложений память выделяется с помощью подсистемы **malloc**.

Подсистема malloc - это API управления памятью, состоящий из следующих функций:

- **malloc**
- **calloc**
- **realloc**
- **free**
- **mallopt**
- **mallinfo**
- **alloca**
- **valloc**
- **posix_memalign**

Подсистема **malloc** управляет объектом логической памяти, который называется *кучей*. Куча - это область памяти в адресном пространстве приложения, расположенная после последнего байта данных, размещенного компилятором. Память кучи выделяется и освобождается с помощью API подсистемы **malloc**.

Подсистема **malloc** выполняет следующие основные операции работы с памятью:

- Выделение:
  Осуществляется функциями **malloc**, **calloc valloc**, **alloca** и **posix_memalign**.
- Освобождение:
  Выполняется функцией **free**.
- Изменение размера:
  Выполняется функцией **realloc**.

Функции **mallopt** и **mallinfo** поддерживаются для совместимости с System V. Функция **mallinfo** может применяться для получения информации о куче, с которой работает функция **malloc**, во время создания программы. С помощью функции **mallopt** можно освобождать память блоками, кратными размеру страницы и выровненными по границе страницы, а также отключить стандартную стратегию выделения памяти. Функция **valloc** аналогична функции **malloc** и поддерживается для совместимости со стандартом Berkeley Compatibility Library.

Дополнительная информация приведена в следующих разделах:

Работа с кучей процесса

Адрес первого байта, следующего после последнего байта инициализированных данных программы, - это адрес символьной переменной **_edata**. Переменная **_edata** указывает на начало кучи процесса, увеличиваемой подсистемой **malloc** при размещении первого блока данных. Подсистема **malloc** расширяет кучу процесса, увеличивая значение brk, обозначающее конец кучи. Для этого вызывается функция **sbrk**. Последующие вызовы функций подсистемы **malloc** расширяют кучу в соответствии с требованиями приложения.

Куча состоит из *занятых* и *свободных* блоков памяти. Память, которую можно выделить приложению, называется пулом свободных блоков. При выделении блока он удаляется из пула свободных блоков, после чего вызывающей функции возвращается указатель на этот блок. При изменении размера памяти выделяется блок другого размера, данные из исходного блока перемещаются в новый блок, а исходный блок освобождается. Выделенные блоки памяти состоят из областей кучи процесса, используемых приложением. Поскольку блоки памяти физически не удаляются из кучи (изменяется только их состояние со свободного на занятый), то при освобождении памяти приложением размер кучи процесса не уменьшается.

Адресное пространство 32-разрядных приложений

Адресное пространство 32-разрядных прикладных программ разделяется в системе на следующие сегменты:

Сегмент	Описание
С `0x00000000` по `0x0fffffff`	Содержит ядро.
С `0x10000000` по `0x1fffffff`	Содержит текст прикладной программы.
С `0x20000000` по `0x2fffffff`	Содержит данные и стек прикладной программы, а также кучу процесса.
С `0x30000000` по `0xcfffffff`	Общая память и память для **mmap**.
С `0xd0000000` по `0xdfffffff`	Содержит текст общих библиотек.
С `0xe0000000` по `0xefffffff`	Общая память и память для **mmap**.
С `0xf0000000` по `0xffffffff`	Содержит данные общей библиотеки приложения.

Адресное пространство процесса 64-разрядных приложений

Адресное пространство 64-разрядных прикладных программ разделяется в системе на следующие сегменты:

Сегмент	Описание
С 0x0000 0000 0000 0000 по 0x0000 0000 0fff ffff	Содержит ядро.
0x0000 0000 f000 0000 по 0x0000 0000 ffff ffff	Зарезервировано.
С 0x0000 0001 0000 0000 по 0x07ff ffff ffff ffff	Содержит текст и данные прикладной программы, кучу процесса, а также общую память и память для **mmap**.
С 0x0800 0000 0000 0000 по 0x08ff ffff ffff ffff	Частные объекты.
С 0x0900 0000 0000 0000 по 0x09ff ffff ffff ffff	Текст и данные общей библиотеки.
С 0x0f00 0000 0000 0000 по 0x0fff ffff ffff ffff	Стек приложения.

Примечание: В AIX выделение памяти приложениям осуществляется способом отложенной подкачки. Когда память выделяется приложению некоторой процедурой (например, **malloc**), пространство подкачки для этой памяти не будет выделено до тех пор, пока приложение не обратится к ней. Этот прием хорошо работает для приложений, выделяющих большие и разбросанные фрагменты памяти. Однако это может сказаться на переносимости приложений, выделяющих очень большие области памяти. Если приложение ожидает, что вызовы malloc возвращают ошибку при недостатке общей памяти, то оно может выделить слишком много памяти. Когда впоследствии происходит обращение к этой памяти, то пространство подкачки системы быстро истощается, и операционная система убивает процессы, чтобы избежать исчерпания виртуальной памяти. При выделении памяти приложение должно убедиться, что для запроса на выделение достаточно общей памяти. Если переменную среды **PSALLOC** задать равной `PSALLOC=early`, то будет применяться алгоритм статического выделения памяти. В этом режиме пространство подкачки выделяется сразу же при поступлении запроса. Дополнительная информация приведена в разделе Пространство подкачки и виртуальная память в *Управление операционной системой и устройствами*.

Описание стратегии выделения памяти в системе

Стратегия выделения памяти связана с набором структур данных и алгоритмов, на основе которых моделируется куча и реализуются операции выделения, освобождения и изменения размера области памяти. Подсистема **malloc** поддерживает следующие стратегии выделения памяти: стратегия по умолчанию, стратегия watson, стратегия malloc 3.1 и пользовательская стратегия. API доступа к подсистеме **malloc** не зависит от стратегии; меняется только реализация стратегии.

Для указания стратегии выделения памяти и опций, включая отладочные, служат следующие переменные среды:

- **MALLOCTYPE** - задает стратегию выделения памяти.
- **MALLOCOPTIONS** - задает опции выбранной стратегии выделения памяти.
- **MALLOCDEBUG** - задает опции отладки выбранной стратегии выделения памяти.
- **MALLOCALIGN** - задает выравнивание **malloc** по умолчанию за пределами программы.

Стандартная стратегия более эффективна для большинства типичных приложений. Ряд особенностей других стратегий может дать выигрыш в некоторых нестандартных ситуациях. Подробные сведения об этом приведены в разделе Сравнение различных стратегий выделения памяти.

Некоторые опции стратегии выделения памяти можно использовать совместно с другими опциями. Для этого при задании переменных среды **MALLOCOPTIONS** и **MALLOCDEBUG** опции следует указывать, разделяя их запятыми.

Переменная среды **MALLOCALIGN** позволяет задать выравнивание по умолчанию, предпочитаемое для каждой операции выделения памяти **malloc()**. Пример:

```
MALLOCALIGN=16;  export MALLOCALIGN
```

В переменной среды **MALLOCALIGN** можно указать любое значение, кратное двум и превышающее размер указателя в соответствующем режиме выполнения или равное ему (4 байта для 32-разрядного режима и 8 байт для 64-разрядного режима). Для 32-разрядных программ с поддержкой векторов в качестве значения

этой переменной можно указать 16, обеспечив тем самым выравнивание всех данных **malloc()**. Обратите внимание, что для 64-разрядных векторных программ память изначально выделяется с 16-байтовым выравниванием.

Кроме того, программа с помощью процедуры **mallopt(M_MALIGN, 16)** можно изменить значение **malloc()** по умолчанию, применив 16-байтовое выравнивание. Процедура **mallopt(M_MALIGN)** позволяет программе управлять операциями выделения памяти malloc по умолчанию в динамическом режиме.

Описание стандартной стратегии выделения памяти

В стратегии выделения памяти по умолчанию свободная память в куче хранится в виде узлов *декартова* бинарного дерева поиска. Узлы в дереве упорядочены слева направо по возрастанию адреса и сверху вниз по размеру (так что потомок не может быть больше по размеру, чем родитель). Такая структура данных не накладывает ограничения на размер блоков, поддерживаемых деревом, позволяя работать с блоками любого размера. Алгоритмы реорганизации дерева позволяют оптимизировать время поиска, вставки и удаления узлов, а также решают проблему фрагментации.

В стандартной стратегии предусмотрены следующие дополнительные возможности:

Выделение

Обслуживание любых операций выделения памяти требует дополнительных затрат. Они обусловлены необходимостью создавать префикс метаданных и выравнивать блоки памяти по границе. Для любой операции размещения размер префикса метаданных составляет 8 и 16 байт для 32- и 64-разрядных программ соответственно. Фрагменты памяти выравниваются по границе 16 или 32 байт. Отсюда общая память, расходуемая при выделении области памяти размером n байт составляет:

```
размер = округление_вверх(n + размер_префикса, выравнивание)
```

Например, для размещения 37 байт 32-разрядной программой потребуется `округление_вверх(37 + 8, 16)`, то есть 48 байт.

Из дерева удаляется узел с наименьшим адресом, размер которого больше либо равен значению размера. Если найденный блок больше запрошенного, его остаток выделяется в отдельный блок. Второй блок *runt* возвращается в дерево свободных блоков. Основная часть блока возвращается инициатору.

Если в дереве свободных блоков нет блока достаточного размера, куча расширяется, и блок, образованный в результате расширения, добавляется в дерево свободных блоков. После этого выполняется описанная выше операция выделения.

Освобождение

Блоки памяти освобождаются с помощью функции **free** и помещаются в корень дерева свободных блоков. Для всех узлов, расположенных в дереве выше позиции вставки нового узла, выполняется проверка, не являются ли они смежными с добавляемым блоком. Если да, то два узла объединяются, и в дерево добавляется узел для объединенного блока. Если смежные блоки не найдены, то узел просто располагается в соответствующей позиции дерева. Объединение смежных блоков позволяет значительно снизить степень фрагментации кучи.

Изменение размера

Если размер запрошенного блока больше размера исходного блока, то исходный блок возвращается в дерево свободных блоков с помощью функции **free**. В результате он объединяется со всеми свободными смежными блоками. После этого выделяется новый блок запрошенного размера, данные копируются из исходного блока в новый, после чего он возвращается инициатору.

Если размер запрошенного блока меньше размера исходного блока, блок разбивается на части, и меньшая из полученных частей возвращается в дерево свободных блоков.

Ограничения

Ниже перечислены доступные опции стратегии выделения памяти по умолчанию:

- Режим с несколькими кучами malloc
- Сегменты malloc
- Опция malloc Disclaim
- Кэш нитей malloc
- Описание опции no_overwrite

Описание стратегии выделения памяти watson

В стратегии выделения памяти Watson свободная память в куче хранится в виде узлов двух независимых красно-черных деревьев. Первое упорядочено по адресу, второе - по размеру. Операции с красно-черными деревьями проще и более эффективны, чем с декартовым деревом стратегии выделения памяти по умолчанию, поэтому стратегия Watson часто работает быстрее, чем стратегия по умолчанию.

Выделение

Дополнительные затраты стратегии выделения памяти Watson такие же, как у стратегии выделения памяти по умолчанию.

В дереве размера ищется блок наименьшего размера, но не меньший, чем размер запрошенной области памяти. Затем этот блок удаляется из дерева размера. Если найденный блок больше запрошенного, то он делится на два блока: второй блок - запрошенного размера, а первый - остаток - выделяется в отдельный блок. Первый блок *runt* возвращается в дерево размера. Второй блок передается инициатору. Если размер блока в дереве размера в точности соответствует запросу, то блок удаляется из обоих деревьев и передается инициатору.

Если в дереве свободных блоков нет блока достаточного размера, куча расширяется, и блок, образованный в результате расширения, добавляется в деревья размера и адресов. После этого выполняется описанная выше операция выделения.

Освобождение

Блоки памяти освобождаются с помощью функции **free** и помещаются в корень дерева адресов. Для всех узлов, расположенных в дереве выше позиции вставки нового узла, выполняется проверка, не являются ли они смежными с добавляемым блоком. Если да, то два узла объединяются, и в дерево размера добавляется узел, соответствующий объединенному блоку. Если смежные блоки не найдены, то узел просто размещается в соответствующей позиции деревьев размера и адресов.

После вставки в обоих деревьях проводится проверка правильности структуры.

Изменение размера

Если размер запрошенного блока больше размера исходного блока, то исходный блок возвращается в деревья свободных блоков с помощью функции **free**. В результате он объединяется со всеми свободными смежными блоками. После этого выделяется новый блок запрошенного размера, данные копируются из исходного блока в новый, после чего он возвращается инициатору.

Если размер запрошенного блока меньше размера исходного блока, то блок разбивается на части, и остаток возвращается в дерево свободных блоков.

Ограничения

Ниже перечислены доступные опции стратегии выделения памяти Watson:

- Режим с несколькими кучами malloc
- Опция malloc Disclaim
- Кэш нитей malloc
- Описание опции no_overwrite

Описание стратегии выделения памяти malloc версии 3.1

Для применения стратегии выделения памяти malloc 3.1 следует перед запуском процесса указать значение `MALLOCTYPE=3.1`. В результате все 32-разрядные программы, запущенные из данной оболочки, будут применять стратегию выделения памяти malloc 3.1 (63-разрядные по-прежнему будут использовать стандартную стратегию).

В стратегии malloc 3.1 куча представляет собой набор из 28 хэш-блоков, каждый из которых указывает на список. Каждый список содержит блоки определенного размера. Индекс хэш-блока определяет размер блоков в связанном с ним списке. Размер блока вычисляется по следующей формуле:

$$size = 2^{i+4}$$

где *i* - это номер хэш-блока. Это означает, что нулевой хэш-блок ссылается на список из блоков 20+4 = 16 байт длиной. Если предположить, что префикс занимает 8 байт, такие блоки могут применяться для запросов на область памяти размером от 0 до 8 байт. В приведенной ниже таблице показана взаимосвязь между размером запрошенной области памяти и хэш-блоками.

Примечание: Для такого алгоритма может потребоваться вдвое больше памяти, чем было запрошено приложением. Для хэш-блока размером более 4096 байт потребуется дополнительная страница, так как данные, объем которых больше либо равен странице, размещаются в блоках памяти размером со страницу. Поскольку префикс расположен непосредственно перед блоком, он займет всю страницу.

Хэш-блок	Размер блока	Диапазон запросов	Число требуемых страниц
0	16	0... 8	
1	32	9... 24	
2	64	25... 56	
3	128	57 ... 120	
4	256	121 ... 248	
5	512	249 ... 504	
6	1 КБ	505 ... 1 КБ - 8	
7	2 КБ	1 КБ - 7 ... 2 КБ - 8	
8	4 КБ	2 КБ - 7 ... 4 КБ - 8	2
9	8 КБ	4 КБ-7 ... 8 КБ - 8	3
10	16 КБ	8 КБ - 7 ... 16 КБ - 8	5
11	32 КБ	16 КБ - 7 ... 32 КБ - 8	9
12	64 КБ	32 КБ - 7 ... 64 КБ - 8	17
13	128 КБ	64 КБ-7 ... 128 КБ-8	33
14	256 КБ	128 КБ-7 ... 256 КБ-8	65
15	512 КБ	256 КБ-7 ... 512 КБ-8	129
16	1 МБ	256 КБ-7 ... 1 МБ-8	257
17	2 МБ	1 МБ-7 ... 2 МБ-8	513
18	4 МБ	2 МБ-7 ... 4 МБ-8	1 КБ + 1
19	8 МБ	4 МБ-7 ... 8 МБ-8	2 КБ + 1

Хэш-блок	Размер блока	Диапазон запросов	Число требуемых страниц
20	16 МБ	8 МБ-7 ... 16 МБ-8	4 КБ + 1
21	32 МБ	16 МБ-7 ... 32 МБ-8	8 КБ + 1
22	64 МБ	32 МБ-7 ... 64 МБ-8	16 КБ + 1
23	128 МБ	64 МБ-7 ... 128 МБ-8	32 КБ + 1
24	256 МБ	128 МБ-7 ... 256 МБ-8	64 КБ + 1
25	512 МБ	256 МБ-7 ... 512 МБ-8	128 КБ + 1
26	1024 МБ	512 МБ-7 ... 1024 МБ-8	256 КБ + 1
27	2048 МБ	1024 МБ-7 ... 2048 МБ-8	512 КБ + 1

Выделение

Перед выделением блока из пула свободных блоков число запрошенных байт преобразуется в индекс массива хэш-блоков по следующей формуле:

```
требуется = запрошено + 8

Если требуется <= 16,
то
bucket = 0

Если требуется > 16,
то
bucket =
(log(требуется)/log(2) округленное в
меньшую сторону до ближайшего целого числа) - 3
```

Размер блоков в списке, на который ссылается хэш-блок, можно вычислить по формуле: `размер блока = 2 хэш-блок + 4`. Если список, на который ссылается хэш-блок, пустой, то в него добавляются блоки путем выделения памяти с помощью функции **sbrk**. Если размер блока меньше страницы, то функция **sbrk** выделяет страницу. При этом число блоков, добавленных в список, равно размеру страницы, поделенному на размер блока. Если размер блока больше либо равен размеру страницы, необходимый объем памяти выделяется с помощью функции **sbrk**, и к списку свободных блоков хэш-блока добавляется всего один блок. Если список свободных блоков не пуст, то инициатору возвращается первый блок списка. При этом указатель на начало списка связывается со следующим блоком.

Освобождение

При освобождении блока памяти, как и при выделении, подсчитывается индекс хэш-блока. После этого освобожденный блок добавляется в начало списка свободных блоков хэш-блока.

Изменение размера

При изменении размера блока требуемый размер блока сравнивается с его текущим размером. Поскольку один и тот же хэш-блок применяется для выделения областей памяти различного размера, новый блок часто относится к тому же хэш-блоку, что и старый. В этом случае длина префикса обновляется в соответствии с новым размером, и инициатору запроса возвращается тот же блок. Если требуется увеличить размер блока, текущий блок освобождается и выделяется новый блок, связанный с другим хэш-блоком. При этом данные копируются из старого блока в новый.

Ограничения

Настройка `MALLOCTYPE=3.1` только включит стратегию malloc 3.1 для 32-разрядных программ. Для того чтобы 64-разрядные программы использовали стратегию malloc 3.1 переменной среды **MALLOCTYPE** должно быть явно задано значение `MALLOCTYPE=3.1_64BIT`. Данная стратегия размещения менее эффективна, чем стандартная, и в большинстве случаев не рекомендуется.

Ниже перечислены доступные опции стратегии выделения памяти malloc 3.1:

- Опция malloc Disclaim
- Описание опции no_overwrite

Описание стратегии выделения пулов

Пул **Malloc** - это высокопроизводительная команда-клиент функций libc **malloc**, **calloc**, **free**, **posix_memalign** и **realloc** для управления объектами хранения меньше, чем 513 байт. Быстродействие достигается за счет значительно более коротких путей и лучшего использования кэша данных. Есть также дополнительное преимущество для приложений со многими нитями. Оно состоит в том, что метки локальных пулов нитей используются вместо атомарных операций. Данную команду-клиент можно использовать вместе с любой из схем управления хранением, предоставляемой в настоящий момент в libc (*yorktown* и *watson*).

Для того чтобы использовать пул **malloc** выполните следующую команду:

```
export MALLOCOPTIONS=pool<:max_size>
```

При указании этой опции при инициализации **malloc** создается набор пулов, в котором каждый пул - это связанный список объектов фиксированного размера. Самый маленький пул может содержать объекты, имеющие размер указателей (например, 8 байт для 32-разрядных приложений или 16 байт для 64-разрядных). Последующие пулы содержит объекты, имеющие размер больше на величину размера указателя. Это означает, что есть 128 пулов для 32-разрядных приложений и 64 пула для 64-разрядных приложений. Набор пулов представлен как массив указателей, которые "отмечают" связанные списки.

Пул **Malloc** использует свою собственную память - кучу пула, которая не используется совместно со стандартным **malloc**. Если указана опция **max_size**, то она округляется до ближайшего большего значения с шагом 2 МБ и используется для управления размером кучи пула. Опцию **max_size** можно указывать как десятичное или шестнадцатеричный число, перед которым указано 0x или 0X (например, `export MALLOCOPTIONS=pool:0x1700000` приравняет **max_size** к 24 МБ после округления.

Для 32-разрядных приложений размер кучи пула начинается с 2 МБ. Если требуется дополнительное пространство для хранения и общий размер хранилища кучи пула меньше, чем **max_size**, то добавляется 2 МБ. Зоны 2 МБ не должны соседствовать друг с другом. Для 64-разрядных приложений размещается одна куча пула, размером **max_size** во время инициализации **malloc**. Ее размер не увеличивается. Если **max_size** не указан, он равен 512 МБ для 32-разрядных приложений и 32 МБ для 64-разрядных. Для 32- и 64-разрядных режимов **max_size** будет равен 512 МБ, если задан больший размер. В 32-разрядном режиме в переменной **max_size** указывается значение 512 МБ, а в 64-разрядном режиме - значение 3.7 ГБ, если задан больший размер.

Использование хранилища

Все метки пула первоначально пусты или равняются NULL. При обработке запроса к пулу **malloc** в случае, если соответствующий пул пуст, вызывается процедура, которая размещает данные кучи пула в смежном куске, размером 1024 байт на границе 1024 байт. "Создаются" множественные объекты требуемого размера. Адрес первого возвращается как ответ на запрос, а остальные объекты соединены вместе и расположены на метке пула. Для каждого куска программы длинной 1024 байт в дополнительный таблице есть запись, размером 2 байта, используемая функцией free для определения размера возвращаемого объекта.

Когда объект освобождается пулом **malloc** он "вталкивается" в соответствующую метку пула. Попыток объединить блоки для создания объектов большего размера не делается.

Вследствие данного поведения, пул **malloc** может использовать больше места для хранения, чем другие формы **malloc**.

Выравнивание

Стандартное выравнивание для функций **malloc()**, **calloc()** и **realloc()** должно быть задано с помощью переменных среды **MALLOCALIGN**. Функция **posix_memalign()** работает, даже если не задана переменная среды **MALLOCALIGN**. Если **MALLOCALIGN** больше 512, пул **malloc** не используется.

Эффективность кэша

Объекты памяти, размещенные в **пуле malloc** не имеют приставок и суффиксов. Линии данных кэша более плотно упакованы данными приложений. Так как размер всех объектов памяти выровнен по значению 2, каждый объект содержится в минимальном количестве линий кэша. Функции **malloc** и **free** не сканируют деревья или связанные списки и, поэтому, не “загрязняют” кэш.

Поддержка нескольких нитей

Пулы **malloc** могут значительно улучшить производительность в сценариях использования многих нитей, так как это уменьшает число конфликтов блокировки и необходимость атомарных операция.

Поддержка распределения нагрузки

В отдельных сценариях с несколькими нитями возможно чрезмерное увеличение размера свободного пула одной из нитей за счет повторяющегося освобождения динамически выделяемой памяти. Однако другие нити не смогут использовать эту память.

Поддержка распределения нагрузки позволяет передать половину памяти из пула нити в глобальный пул после достижения заданного порогового значения. Пороговые значения для перераспределения памяти пулов нитей можно настроить.

Для включения поддержки распределения нагрузки необходимо экспортировать следующие параметры:

```
export MALLOCOPTIONS=pool:0x80000000,pool_balanced
```

```
export MALLOCFREEPOOL=min_size<-max_size>:threshold_value<,min_size<-max_size>:
threshold_value, ... >,default:threshold
```

В следующем примере задается пороговое значение 256 байт для пулов с блоками размером 0 - 16 байт, а также пороговое значение 512 байт для пулов с блоками размером 32 байт. Для остальных пулов применяется пороговое значение 128 байт.

```
export MALLOCFREEPOOL=0-16:256,32:512,default:128
```

Поддержка отладки

Не существует версии отладки данной высокопроизводительного команды-клиента. Если задана переменная среды **MALLOCDEBUG**, опция пула игнорируется. Подразумевается, что приложения будут отлаживаться с помощью "нормального" **malloc** до активации использования пула.

Описание пользовательской стратегии выделения памяти

В подсистеме malloc предусмотрены механизмы, посредством которых пользователи могут применять собственные алгоритмы управления кучей и выделением памяти.

Описание опции no_overwrite

Дополнительная опция `no_overwrite` применима ко всем стратегиям выделения памяти. Для сокращения числа глобальных связей дескрипторы функций подсистемы **malloc** заменяются дескрипторами базовых функций, фактически выполняющих соответствующие операции. Так как некоторые программы, например, отладчики сторонних производителей, не всегда правильно работают с измененными указателями, для отключения этой функции оптимизации предусмотрена опция `no_overwrite`.

Для отключения этой функции оптимизации задайте перед запуском процесса `MALLOCOPTIONS=no_overwrite`.

Сравнение различных стратегий выделения памяти

Приведенная выше информация о стратегиях выделения памяти предоставляют разработчикам широкие возможности использования этих стратегий как по отдельности, так и в сочетании друг с другом. Разработчик должен сам определить оптимальные параметры стратегии выделения памяти согласно потребностям приложения.

Различия между стандартной стратегией и стратегией malloc 3.1

Основной недостаток стратегии malloc 3.1 заключается в том, что при выделении памяти размер запрошенной области округляется в большую сторону до ближайшего числа, являющегося степенью двойки. Это приводит к существенной фрагментации виртуальной и оперативной памяти и некомпактному размещению ссылок. В стандартной стратегии отводится ровно столько памяти, сколько было запрошено, и применяются усовершенствованные средства управления освободившимися областями памяти.

К сожалению, некоторые программы используют особенности реализации стратегии malloc 3.1, и поэтому применение другой стратегии может снизить их производительность или даже нарушить нормальную работу. Например, программа, в которой возникает выход за границы массива, может нормально работать со стратегией malloc 3.1 потому, что в результате округления память под этот массив будет выделена с запасом. Применение стандартной стратегии памяти с большой вероятностью нарушит работу такой программы, поскольку под массив будет выделяться ровно столько памяти, сколько запрошено.

Рассмотрим другой пример. Из-за особенностей алгоритма управления освободившимися блоками памяти в стратегии malloc 3.1 программы почти всегда получают память, в которой всем ячейкам присвоены нулевые значения (страница обнуляется при первом обращении процесса к ней). Работа некоторых программ может зависеть от этого побочного эффекта. В действительности функция **malloc** не обязана обнулять память. Эта операция только снижает производительность программ, которые выполняют инициализацию памяти самостоятельно. Поскольку стандартная стратегия более активно использует высвободившиеся области памяти, она может нарушить работу программ, рассчитывающих на обнуление выделяемой памяти со стороны функции **malloc**.

Если программа регулярно увеличивает размер области памяти, выделенной для структуры данных, то в стратегии malloc 3.1 с меньшей вероятностью потребуется перемещать эту структуру. Во многих случаях функции **realloc** будет достаточно "избыточной" памяти, которая была выделена с самого начала за счет округления запрошенного размера области памяти в алгоритме malloc 3.1. В стандартной стратегии в таких случаях структура данных почти всегда перемещается, потому что с большой вероятностью вся память вокруг увеличиваемой области будет занята. В описанном случае функция **realloc** будет выполняться быстрее, если применяется стратегия malloc 3.1, а не стандартная стратегия. Однако такой выигрыш обеспечивается только за счет недостатков в реализации программы.

Отладка ошибок приложений в управлении кучей

В подсистеме **malloc** предусмотрены средства отладки, помогающие разработчику исправить ошибки приложений в управлении кучей. Эти средства отладки задаются переменной среды **MALLOCDEBUG**.

Сводная таблица переменных среды и параметров malloc

В следующей таблице приведены данные по совместимости переменных среды **MALLOCTYPE** и **MALLOCOPTIONS**.

Таблица 80. Совместимость переменных среды **MALLOCTYPE** *и* **MALLOCOPTIONS**

	Несколько куч (и связанные опции)	**сегменты (и связанные опции)**	**Кэш нитей**	**disclaim**	**no_overwrite**
Распределитель по умолчанию	да	да	да	да	да
3.1	нет	нет	да	да	да
Watson	нет	нет	нет	нет	нет
Watson2	нет	нет	нет	нет	нет
Пользователь:	нет	нет	нет	нет	да

Таблица 81. Совместимость переменных среды **MALLOCDEBUG** *и* **MALLOCTYPE**

	<Распределитель по умолчанию> York Town	**3.1**	**Watson**	**Watson2**	**Пользователь:**
catch_overflow (и связанные опции)	да	нет	да	да	нет
report_allocations	да	нет	да	да	нет
postfree_checking	да	нет	да	да	нет
validate_ptrs	да	нет	да	да	нет
трассировка	да	нет	да	да	нет
log	да	нет	да	да	нет
verbose	нет	нет	нет	нет	нет

Все опции All **MALLOCDEBUG** поддерживаются и совместимы с **MALLOCOPTIONS**.

Описание стратегии выделения памяти Watson2

Подсистема malloc Watson2 адаптируется к поведению приложения, когда оно переходит от использования одной нити к использованию многих нитей и наоборот. Она применяет механизм, специфичный для работы с нитями, который использует переменное число структур кучи, которое зависит от поведения программы. Поэтому, опции конфигурации не требуются. Подсистема malloc Watson 2 имеет сниженную (logN) стоимость операции для многих рабочих нагрузок, так как огромное число операций может быть выполнено за постоянное время без синхронизации.

Выделение

Выделение управляется посредством комбинации механизмов. Эти механизмы зависят от параметров, таких как число активных нитей, размер запроса и хронология освобождения ресурсов для процесса. Набор механизмов происходит от кэширования для нитей и использует переменное число куч, имеющих привязку нити к структуре Double-red-black и объединение на основе страниц.

Освобождение

Освобождение зависит от тех же параметров, что и поведение выделения. Обычно, возвращающий блок захватывается в связанный с нитью кэш. На основании привязки кучи и использования памяти, память может быть возвращена одной из нескольких структур кучи. Иногда, содержимое нескольких структур кучи объединяется в общую структуру кучи для повышения консолидации и сокращения фрагментации кучи. Для повышения устойчивости к ошибкам приложения, распределитель определяет освобождение неверных указателей или поврежденных блоков до некоторой степени и отфильтровывает эти операции.

Изменение размера

Большие блоки памяти, которые достаточны, повторно используются. Если текущий блок не может удовлетворить запрос, он заменяется с помощью обычного освобождения и выделения.

Ограничения

Подсистема malloc Watson2 является адаптивной к приложениям и не требует дополнительных опций, но подсистема malloc Watson2 поддерживает следующие функции отладки, управляемые переменной `MALLOCDEBUG`: `validate_ptrs`, `report_allocations` и `trace`. Относящиеся к выделениям отчеты могут быть перенаправлены в файл с помощью опции `output:<имя_файла>`. Более подробная информация о переменной `MALLOCDEBUG` находится в разделе “Отладчик malloc” на стр. 637.

Понятия, связанные с данным:

“Адресное пространство программы - обзор” на стр. 600
В состав базовой операционной системы включен набор служб распределения памяти между приложениями.

“Отладчик malloc” на стр. 637
Отладка приложений, в которых возникают проблемы с распределением памяти с помощью подсистемы **malloc** очень часто требует больших усилий и не всегда дает желаемые результаты. Это связано с тем, что чаще всего возникновение самой ошибки и ее видимое проявление разнесены по времени.

“Пользовательские аналоги функций из подсистемы памяти”
Пользователи могут заменить функции подсистемы памяти (функции **malloc**, **calloc**, **realloc**, **free**, **mallopt** и **mallinfo** subroutines) собственными функциями.

“Режим с несколькими кучами malloc” на стр. 644
По умолчанию подсистема malloc использует одну кучу (пул свободной памяти).

“Сегменты malloc” на стр. 645
Наборы функции malloc представляют собой дополнительные расширения стандартной функции распределения памяти на основе наборов.

“Функция трассировки malloc” на стр. 649
Функция трассировки malloc - это дополнительная функция подсистемы malloc, применяемая вместе с трассировщиком.

“Протокол malloc” на стр. 650
Протокол malloc - это дополнительное расширение подсистемы **malloc**, позволяющее пользователю получать информацию об активных областях, выделенных вызывающим процессом. Эти данные могут применяться для определения причин возникновения неполадок и анализа производительности.

“Опция malloc disclaim” на стр. 651
Функция отказа от памяти malloc - дополнительное расширение подсистемы malloc, позволяющее пользователю автоматически отказываться от памяти, возвращаемой функцией **free**.

Пользовательские аналоги функций из подсистемы памяти

Пользователи могут заменить функции подсистемы памяти (функции **malloc**, **calloc**, **realloc**, **free**, **mallopt** и **mallinfo** subroutines) собственными функциями.

Примечание: Пользовательские подсистемы памяти, написанные на языке C++, не поддерживаются, так как в библиотеке C++ **libC.a** используется подсистема памяти **libc.a**.

Существующая подсистема памяти может применяться всеми приложениями, независимо от того, используются ли в них нити. Пользовательская подсистема памяти должна обеспечивать поддержку нитей, для того чтобы она могла применяться как в процессах с нитями, так и в процессах без нитей. Наличие такой поддержки не проверяется, однако загрузка модуля памяти без поддержки нитей в приложение с нитями может привести к повреждению памяти и данных.

32- и 64-разрядные объекты пользовательской подсистемы памяти должны размещаться в архиве вместе с 32-разрядным общим объектом `mem32.o` и 64-разрядным общим объектом `mem64.o`.

Пользовательские общие объекты должны экспортировать следующие идентификаторы:

- __malloc__
- __free__

- __realloc__
- __calloc__
- __mallinfo__
- __mallopt__
- __malloc_init__
- __malloc_prefork_lock__
- __malloc_postfork_unlock__

Дополнительно такие объекты могут экспортировать следующий идентификатор:
- __malloc_start__
- __posix_memalign__

Если данные символы не выйдут, выполнение программы продолжится.

Ниже приведены определения пользовательских функций:

void *__malloc__(size_t) :
Пользовательский эквивалент функции **malloc**.

void __free__(void *) :
Пользовательский эквивалент функции **free**.

void *__realloc__(void *, size_t) :
Пользовательский эквивалент функции **realloc**.

void *__calloc__(size_t, size_t) :
Пользовательский эквивалент функции **calloc**.

int __mallopt__(int, int) :
Пользовательский эквивалент функции **mallopt**.

struct mallinfo __mallinfo__() :
Пользовательский эквивалент функции **mallinfo**.

void __malloc_start__()
Эта функция вызывается один раз перед вызовом любой другой пользовательской функции malloc.

void __posix_memalign__()
Пользовательский эквивалент функции **posix_memalign**. Если данный символ не выйдет, программа будет продолжаться, но вызов процедуры **posix_memalign** может вызвать ошибку.

Ниже перечислены функции, применяемые подсистемой с нитями для управления пользовательской подсистемой памяти в среде с несколькими нитями. Они вызываются только в том случае, если приложение и/или пользовательский модуль связаны с **libpthreads.a**. Эти функции должны быть определены и экспортированы даже в том случае, если пользовательская подсистема не поддерживает нити и не связана с **libpthreads.a**. В противном случае объект не будет загружен.

void __malloc_init__(void)
Вызывается процедурой инициализации API нитей. Данная функция применяется для инициализации пользовательской подсистемы памяти с поддержкой нитей. В большинстве случаев данная функция создает и инициализирует некоторые типы блокировок данных. Даже если модуль пользовательской подсистемы памяти связан с **libpthreads.a**, пользовательская подсистема памяти *должна* правильно работать еще до того, как будет вызвана функция `__malloc_init__()`.

void __malloc_prefork_lock__(void)
Вызывается API нитей при обращении к функции `fork`. Данная функция контролирует, что перед вызовом `fork()` подсистема находится в допустимом состоянии и остается в нем до завершения функции `fork()`. В большинстве случаев эта функция устанавливает блокировки подсистемы памяти.

void __malloc_postfork_unlock__(void)

Вызывается API нитей при обращении к функции fork. Эта функция разблокирует подсистему памяти для родительского и дочернего процесса после выполнения функции fork. Она отменяет действие функции __malloc_prefork_lock__. В большинстве случаев, эта функция снимает блокировки подсистемы памяти.

Все эти функции должны быть экспортированы из общего модуля. В архиве должны быть предусмотрены отдельные модули для 32-разрядных и 64-разрядных функций. Например:

- Модуль **mem.exp**:

```
__malloc__
__free__
__realloc__
__calloc__
__mallopt__
__mallinfo__
__malloc_init__
__malloc_prefork_lock__
__malloc_postfork_unlock__
__malloc_start__
```

- Модуль **mem_functions32.o**:

  Содержит все необходимые 32-разрядные функции

- Модуль **mem_functions64.o**:

  Содержит все необходимые 64-разрядные функции

Ниже приведены примеры создания общих объектов. Параметр -lpthreads следует указывать только в том случае, если объект применяет функции библиотеки pthread.

- Создание 32-разрядного общего объекта:

```
ld -b32 -m -o mem32.o mem_functions32.o \
-bE:mem.exp \
-bM:SRE -lpthreads -lc
```

- Создание 64-разрядного общего объекта:

```
ld -b64 -m -o mem64.o mem_functions64.o \
-bE:mem.exp \
-bM:SRE -lpthreads -lc
```

- Создание архива (32-разрядный общий объект должен называться mem32.o, а 64-разрядный - mem64.o):

```
ar -X32_64 -r архив mem32.o mem64.o
```

Применение пользовательской подсистемы памяти

Пользовательскую подсистему памяти можно подключить различными способами:

- С помощью переменной среды **MALLOCTYPE**
- С помощью глобальной переменной _malloc_user_defined_name в пользовательском приложении

Для применения переменной среды **MALLOCTYPE** нужно задать архив, содержащий пользовательскую подсистему памяти, присвоив переменной **MALLOCTYPE** значение user:*имя-архива*. При этом *имя-архива* должно быть указано в параметре libpath приложения или в переменной среды **LIBPATH**.

Для применения глобальной переменной _malloc_user_defined_name она должна быть объявлена в программе следующим образом:

```
char *_malloc_user_defined_name="архив"
```

архив должен быть указан в переменной libpath программы или в переменной среды **LIBPATH**.

Примечание:

1. При выполнении приложения setuid переменная среды **LIBPATH** игнорируется, поэтому имя архива должно быть определено в параметре libpath приложения.
2. *Имя-архива* не должно содержать информацию о пути.
3. Если *архив* указан и в переменной среды **MALLOCTYPE**, и в глобальной переменной `_malloc_user_defined_name`, то будет применяться архив, указанный в переменной **MALLOCTYPE**.

Информация о 32-разрядных и 64-разрядных функциях

Если архив не содержит и 32-разрядный, и 64-разрядный общий объект, и пользовательская подсистема памяти была подключена с помощью переменной среды **MALLOCTYPE**, то при выполнении 64-разрядных процессов в 32-разрядных приложениях или 32-разрядных процессов в 64-разрядных приложениях может возникнуть сбой. При вызове функции **exec** создается новый процесс, который наследует среду родительского приложения. Это означает, что будет унаследована переменная среды **MALLOCTYPE**, и новый процесс попытается загрузить пользовательскую подсистему памяти. Если в архиве нет объекта для программы данного типа, то подсистема не будет загружена, и работа процесса будет завершена.

Рекомендации по работе с нитями

Все пользовательские функции должны правильно работать в среде с несколькими нитями. Даже если модуль подключен с помощью libpthreads.a, функция `__malloc__()` *должна правильно работать* до того, как будет вызвана функция `__malloc_init__()` и выполнена инициализация библиотеки pthread. Это требование связано с тем, что при инициализации API нитей функция `malloc()` вызывается раньше, чем функция `__malloc_init__()`.

Все подключаемые функции управления памятью должны поддерживать как приложения с нитями, так и приложения без нитей. Функция `__malloc__()` не должна зависеть от функции `__malloc_init__()` (т. е. при выполнении `__malloc__()` должно предполагаться, что функция `__malloc_init__()` еще *не* выполнена.) После выполнения функции `__malloc_init__()` ее результаты могут использоваться функцией `__malloc__()`. Это связано с тем, что при инициализации API нитей функция `malloc()` вызывается перед обращением к функции `__malloc_init__()`.

Следующие переменные позволяют избежать лишних вызовов функций, связанных с нитями:

- Значение переменной `__multi_threaded` равно нулю до тех пор, пока не будет создана первая нить. После создания нити значение становится ненулевым и не обнуляется до конца выполнения процесса.
- Значение переменной `__n_pthreads`, равно `-1` до тех пор, пока не будут инициализированы API нитей; после инициализации значение становится равным 1. С этого момента данная переменная используется в качестве счетчика активных нитей.

Пример:

Ниже приведен фрагмент кода для случая, когда функция `__malloc__()` применяет функцию `pthread_mutex_lock()`:

```
if (__multi_threaded)
pthread_mutex_lock(mutexptr);

/* ..... работа ....... */

if (__multi_threaded)
pthread_mutex_unlock(mutexptr);
```

В этом примере функция `__malloc__()` не применяет API нитей до тех пор, пока не будет завершена их инициализация. Производительность однонитевых приложений также повышается, так как блокировка устанавливается только при запуске второй нити.

Ограничения

Подсистемы памяти, написанные на языке C++, не поддерживаются, так как в них применяются библиотека **libC.a** и подсистема памяти **libc.a**.

Сообщения об ошибках не переводятся, поскольку при инициализации локалей с помощью **setlocale** применяется функция `malloc()`. Если при вызове `malloc()` произойдет сбой, то функция **setlocale** не будет выполнена, и приложение по-прежнему будет применять локаль `POSIX`. Это означает, что будут выводиться только сообщения на английском языке.

Существующие статически скомпонованные программы не могут применять пользовательскую подсистему памяти без повторной компиляции.

Сообщения об ошибках

При первом обращении к функции **malloc** загружается 32- или 64-разрядный объект из архива, заданного в переменной среды **MALLOCTYPE**. Если во время загрузки возникает ошибка, то выводится соответствующее сообщение, и работа приложения завершается. В противном случае проверяется наличие всех необходимых идентификаторов. Если какие-либо идентификаторы отсутствуют, работа приложения завершается, и выводится сообщение со списком недостающих идентификаторов.

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Отладчик malloc

Отладка приложений, в которых возникают проблемы с распределением памяти с помощью подсистемы **malloc** очень часто требует больших усилий и не всегда дает желаемые результаты. Это связано с тем, что чаще всего возникновение самой ошибки и ее видимое проявление разнесены по времени.

Сложность процесса выделения областей памяти еще более усугубляет проблему: одновременно выделяются и освобождаются тысячи областей, доступ к которым осуществляется одновременно и часто асинхронно, причем все это происходит в многонитевой среде, в которой требуется надежная и эффективная синхронизация.

Именно из-за сложности среды наши отладочные средства ориентированы на как можно более раннее обнаружение возникающих ошибок. В этом случае разработчику проще будет найти раздел исходного кода, в котором содержится ошибка.

Для работы с malloc существует множество отладочных средств. Некоторые из них могут работать в сочетании с другими инструментами отладки и с любыми стратегиями выделения памяти, другие же выполняют более специфические задачи. Многие отладочные средства требуют для своей работы дополнительных ресурсов, помимо тех, что выделены для самого процесса. Сам разработчик решает, когда необходимо выделить эти дополнительные ресурсы.

Производительность

Средства отладки не рассчитаны на постоянное применение. Хотя они спроектированы таким образом, чтобы минимизировать снижение производительности системы, такое снижение при широком применении средств отладки все же может быть весьма существенным. В частности, не рекомендуется задавать переменную `MALLOCDEBUG=catch_overflow` в файле **/etc/environment**, потому что скорее всего стабильная работа системы будет нарушена (например, из-за постоянных обращений к пространству подкачки). Средства отладки следует применять только в тех случаях, когда это действительно необходимо.

Поскольку работа отладчика заключается в проведении постоянных проверок в ходе работы программы, применение средств отладки снизит производительность подсистемы **malloc**, но не до такой степени, что дальнейшая работа приложений будет невозможна. После устранения ошибки средства отладки лучше выключить для восстановления нормальной производительности подсистемы **malloc**.

Дисковая и оперативная память

При включении средств отладки или задании переменной `catch_overflow` подсистеме **malloc** требуется существенно больше памяти для работы.

При задании переменной `catch_overflow` каждый запрос malloc увеличивается на длину unsigned long, помноженную на 4096 + 2, а затем округляется до ближайшего большего значения, кратного PAGESIZE. Хотя `catch_overflow` может потребовать слишком много памяти при работе с очень большими программами, отладка большинства приложений не сильно скажется на расходе памяти. При отладке больших приложений рекомендуется указать опции **debug_range** и **functionset** для `catch_overflow`. Это заметно снизит расход памяти, поскольку программа будет отлаживаться по частям.

Каждое выделение памяти процессом записывается в протокол отладки. Если ограничить число сохраняемых указателей стека, то эта дополнительная нагрузка на память может быть снижена.

Если отлаживаемая программа часто вызывает функции выделения памяти подсистемы **malloc**, то при включенных средствах отладки это может привести к серьезному расходу памяти и сделать невозможным нормальное выполнение программы в пределах одного сегмента. В таких случаях рекомендуется увеличить объем памяти, доступный программе, с помощью команды **ulimit** или команды **ld** с опцией **-bmaxdata**.

При выполнении программ с отладчиком следует задать ограничения ulimit для данных (-d) и стека (-s) следующим образом:

```
ulimit -d unlimited
ulimit -s unlimited
```

Для резервирования всех 8 сегментов для 32-разрядного процесса укажите для опции **-bmaxdata** значение `-bmaxdata:0x80000000`.

После отключения отладчика можно восстановить исходные значения ulimit и -bmaxdata.

Команда **ulimit** и опция **-bmaxdata** подробно описаны в разделе Поддержка больших программ.

В некоторых случаях применение средств отладки не оправдано. Поскольку отладчик отводит как минимум отдельную страницу памяти для каждого вызова malloc(), расход памяти у программ, выделяющих множество небольших участков памяти, может очень сильно возрасти. Недостаток памяти или пространства подкачки при выполнении таких программ может привести к новым сбоям. Эти сбои не имеют отношения ни к программам как таковым, ни к средствам отладки malloc.

Примером такой программы может служить X-сервер, выполняющий огромное количество запросов на выделение очень маленьких участков памяти во время инициализации и в ходе дальнейшей работы. Попытка запустить X-сервер (с помощью команды X или xinit) с включенной переменной catch_overflow неминуемо приведет к аварийному завершению работы этого сервера из-за недостатка памяти. Однако указание опций debug_range или functionset позволит осуществить отладку X-сервера по частям. В то же время большинство клиентов X можно без проблем запускать с включенной переменной catch_overflow. Для запуска клиента X с включенной переменной catch_overflow выполните следующие действия:

1. Запустите X-сервер, не задавая переменную catch_overflow.
2. Откройте окно терминала (например, dtterm, xterm или aixterm).
3. Задайте нужные переменные среды в окне терминала и включите catch_overflow.
4. Запустите клиент X из этого окна терминала.

Включение отладчика malloc

По умолчанию функция отладки выделения памяти выключена, но ее можно включить и настроить, задав соответствующее значение переменной среды **MALLOCDEBUG**. Если требуется указать несколько параметров, их можно перечислить через запятую (,). Опции, которые нужно указывать совместно, должны быть совместимы друг с другом.

Примечание: Для выключения отладчика malloc сбросьте переменную среды MALLOCDEBUG с помощью команды **unset MALLOCDEBUG**.

Средства отладки выделения памяти

Средства отладки выделения памяти включают следующие утилиты:

- Обнаружение переполнения буфера
  - align
  - override_signal_handling
  - debug_range
  - functionset
  - allow_overreading
  - postfree_checking
- Функция трассировки malloc
- Протокол malloc
  - report_allocations
  - validate_ptrs
- Обнаружение malloc
  - verbose
  - checkarena
  - output
  - continue
- Malloc debug Fill

Обнаружение переполнения буфера

Ошибки управления памятью иногда бывают связаны с тем, что программа пытается записать больше данных, чем позволяет размер выделенного буфера. Такая операция не влечет за собой немедленных последствий, и ошибка возникает только на этапе, когда потребуются данные, которые хранились в ошибочно занятой (и часто принадлежащей другой программе) области памяти.

Опция отладки `catch_overflow` позволяет отслеживать ситуации ошибочной совместной записи и чтения одних и тех же областей памяти, а также операции повторного освобождения и выделения одних и тех же областей памяти с помощью функции **malloc**. Если отладчик обнаруживает ошибку, то выполняется функция abort или передается сигнал нарушения сегментации **SIGSEGV**. Как правило, при обнаружении ошибки приложение немедленно останавливается, и создается файл core.

Опция `catch_overflow` влияет на следующие стратегии и параметры выделения памяти:

- Стратегия выделения памяти по умолчанию
- Стратегия выделения памяти Watson
- Опция malloc нескольких куч
- Опция malloc кэша нитей
- Опция malloc Disclaim

Опцию отладки `catch_overflow` можно включить, указав значение переменной среды `MALLOCDEBUG=catch_overflow`. При этом будет включено обнаружение чтения и записи за пределами выделенного буфера памяти.

align

По умолчанию функция **malloc** выравнивает возвращаемый указатель по границе ячейки размером в 2 слова. Это требуется для соответствия стандартам, а также для поддержки программ, которые не могут работать без выравнивания (например, использующим компоненты DCE). Однако вследствие ошибки в реализации опции `catch_overflow` программа может записать данные, выходящие за пределы буфера, на величину меньшую, чем значение выравнивания по границе. Такие ошибки не обнаруживаются с помощью `catch_overflow`. В связи с этим с помощью опции `align` подсистемы **malloc** можно изменить выравнивание по умолчанию, чтобы уменьшить или вообще свести к нулю число байт, которое может быть записано за пределы буфера, не будучи при этом обнаружено. Такое нестандартное выравнивание должно быть указано как степень двойки в пределах от 0 до 4096 включительно (например, 0,1,2,4,...). Значения 0 и 1 интерпретируются одинаково и означают отсутствие выравнивания, то есть любое обращение к памяти за границами выделенной области приведет к нарушению сегментации (SEGFAULT).

Опция `align` действует только при включенной опции `catch_overflow` и в противном случае игнорируется. Для включения нестандартного выравнивания задайте переменную среды **MALLOCDEBUG** следующим образом:

```
MALLOCDEBUG=catch_overflow,align:n
```

где *n* - показатель выравнивания.

Следующая формула позволяет определить, какой критерий будет применяться при выделении памяти при включенной опции `catch_overflow`. Здесь размер - это размер выделяемой области памяти в байтах, а n - показатель выравнивания:

```
((((размер / n) + 1) * n) - размер) % n
```

Следующий пример иллюстрирует зависимость между значением опции align и тем, насколько далеко за границу выделенной области памяти разрешается заходить программе при включенной опции catch_overflow. В следующем примере опции align присвоено значение 2:

```
MALLOCDEBUG=align:2,catch_overflow
```

При включенной опции `catch_overflow` выход за границу выделенной области будет обрабатываться следующим образом:

- Если в запросе на выделение памяти указано четное количество байт, malloc выделит ровно столько байт, сколько указано в запросе. В этом случае приложению будет запрещено выходить за границу выделенной области.
- Если в запросе будет указано нечетное количество байт, то размер выделенной области будет на 1 байт больше запрошенного для обеспечения выравнивания. Приложение сможет выходить за границу исходной области на 1 байт.

override_signal_handling

При включенной опции `catch_overflow` могут возникать следующие виды ошибок:

- Ошибки при обращении к памяти (например, попытки чтения или записи за границами выделенной памяти) - в этом случае выдается сигнал нарушения сегментации (SIGSEGV) и создается файл core.
- Прочие ошибки (например, попытки повторно освободить область памяти) - в этом случае выдается сообщение об ошибке и вызывается функция abort, которая завершает процесс с помощью сигнала SIGIOT.

Если вызывающая программа блокирует или перехватывает сигналы SIGSEGV и SIGIOT, то сообщение об ошибке выдано не будет. Для таких случаев предусмотрена опция override_signal_handling, которая позволяет исправить эту ситуацию без перекомпиляции программы.

Если в добавление к опции catch_overflow указана опция override_signal_handling, то при каждом вызове функций подсистемы **malloc** будут выполнены следующие действия:

1. Отключаются все обработчики сигналов программы для SIGIOT и SIGSEGV.
2. Задается стандартный обработчик сигналов SIGIOT и SIGSEGV (обработчик SIG_DFL).
3. Сигналы SIGIOT и SIGSEGV разблокируются.

Если обработчик сигналов приложения изменит действие для сигнала SIGSEGV между вызовами функций распределения памяти, а затем выполнит недопустимую операцию обращения к памяти, то сообщение об ошибке не будет выдано и при включенной опции $^{\text{catch\_overflow}}$, при этом приложение не будет завершено, а файл дампа не будет создан.

Примечание:

1. Опция override_signal_handling неэффективна при работе с приложениями с несколькими нитями, поскольку в этом режиме используется функция **sigprocmask**, а многие многонитевые приложения пользуются функцией **pthread_sigmask**.
2. Если нить программы вызывает функцию **sigwait**, не включая SIGSEGV и SIGIOT в набор сигналов, и в режиме catch_overflow отладчик обнаружит ошибку в такой нити, то нить зависнет, поскольку в этом режиме могут генерироваться только сигналы SIGSEGV и SIGIOT.
3. Если в функцию ядра будет передан недопустимый указатель, то в ней возникнет ошибка. Как правило, в таких ситуациях функции ядра возвращают errno=EFAULT. Если программа не проверяет коды возврата системных вызовов, то такие ошибки могут проходить незамеченными.

debug_range

По умолчанию при включенной опции catch_overflow проверка выхода за границы области памяти выполняется для каждой операции выделения памяти. Если при этом указана также опция debug_range , то будут обработаны только запросы на выделение памяти, попадающие между заданными пользователем минимальным и максимальным значениями размера. В противном случае проверка выхода за границы выделения памяти проводиться не будет. Данная опция позволяет контролировать использование дополнительных ресурсов памяти для режима catch_overflow, указывая диапазон применимости режима.

Опция debug_range действует только при включенной опции catch_overflow. Для того чтобы ее включить, укажите:

```
MALLOCDEBUG=catch_overflow,debug_range:мин:макс
```

где *мин* - минимальное и *макс* - максимальное значение диапазона, в котором требуется контролировать выход за границы области памяти. Если минимальное значение равно 0, то контроль будет применяться для всех запросов, меньших максимального значения. Если максимальное значение равно 0, то контроль будет применяться для всех запросов, больших минимального значения.

Ограничения

В силу особенностей реализации каждая выделенная область памяти занимает как минимум одну страницу. В связи с этим опция debug_range позволяет снизить затраты памяти, связанные с включением опции catch_overflow, но не устраняет их совсем.

Если функция **realloc** вызывается с запросом на выделение памяти в указанном диапазоне значений, то проверка выхода за границы области памяти будет проводиться даже в том случае, если предыдущий запрос не попадал в этот диапазон. Обратное также справедливо.

Примечание: Если опция override_signal указана вместе с опцией debug_range, то сигналы SIGIOT и SIGSEGV переопределяются при всех операциях выделения памяти.

functionset

В силу особенностей реализации каждая выделенная область памяти занимает как минимум одну страницу. В связи с этим опция functionset позволяет снизить затраты памяти, связанные с включением опции catch_overflow, но не устраняет их совсем.

Если функция **realloc** вызывается из функции, указанной в списке, то проверка выхода за границы области памяти будет проводиться даже в том случае, если предыдущий запрос выполнялся функцией, не входящей в этот список. Обратное также справедливо.

Примечание: Если опция override_signal указана вместе с опцией functionset, то сигналы SIGIOT и SIGSEGV переопределяются при всех операциях выделения памяти.

При сохранении списка функций опции functionset не проверяется, существуют ли эти функции.

allow_overreading

Если программа пытается прочитать данные из области, лежащей за концом выделенного участка памяти, то по умолчанию при включенной опции catch_overflow происходит ошибка нарушения сегментации и создается файл core. Однако пользователь может включить опцию catch_overflow, чтобы обнаруживать более опасные ошибки, чем просто выход за границы выделенного участка памяти. Если указать опцию allow_overreading, то при включенной опции catch_overflow выход за границы выделенного участка памяти будет игнорироваться, и можно будет обнаружить другие, более опасные ошибки.

Опция allow_overreading действует только при включенной опции catch_overflow. Для того чтобы ее включить, укажите:

```
MALLOCDEBUG=catch_overflow,allow_overreading,
```

postfree_checking

Ограничения

Опция postfree_checking расходует очень много памяти. Для программ с большими требованиями к памяти применение опции postfree_checking может оказаться невозможным.

Функция трассировки malloc

Отладочная опция трассировки malloc позволяет осуществлять трассировку вызова функций подсистемы **malloc** с помощью системного трассировщика.

Протокол malloc

Отладочная опция протокола malloc служит для получения информации об активных областях памяти, выделенных подсистемой **malloc**.

report_allocations

Опция report_allocations служит для обнаружения утечек памяти в программе. Опция report_allocations использует информацию из базы данных трассировки malloc для получения сведений об активных областях памяти, используемых программой. Каждое выделение памяти заносится в протокол malloc. При освобождении выделенного участка памяти запись о нем удаляется из базы данных протокола

malloc. При завершении процесса список активных областей памяти выводится в stderr, и в нем перечисляются области памяти, которые были выделены какими-либо процедурами, но не были освобождены.

Опция `report_allocations` работает только при включенном протоколе malloc. Поэтому при включении опции `report_allocations` протокол malloc включается автоматически. Для того чтобы включить опцию `report_allocations`, укажите:

```
MALLOCDEBUG=report_allocations
```

validate_ptrs

По умолчанию функции подсистемы **malloc** не проверяют передаваемые им указатели на адресацию допустимой (т.е. выделенной ранее) области памяти. Если какой-то из указателей неверен, может произойти серьезный сбой кучи. При включенной опции `validate_ptrs` функции подсистемы **malloc** будут проверять передаваемые им указатели на допустимость. Если указатель окажется недопустимым (например, будет адресовать область данных, которая не выделялась ранее с помощью **malloc**), то будет показано сообщение об ошибке с указанием причин ошибки, а затем будет вызвана функция abort и создан файл дампа. Опция `validate_ptrs` работает как опция `verbose`. Опция `validate_ptrs` игнорируется, если включена опция `postfree_checking`.

Для того чтобы включить опцию `validate_ptrs`, укажите:

```
MALLOCDEBUG=validate_ptrs
```

Обнаружение malloc

Отладочная опция обнаружения malloc позволяет находить ошибки во внутренних структурах данных подсистемы **malloc** для каждого вызова функций подсистемы **malloc**.

verbose

Подопция опции обнаружения malloc.

checkarena

Подопция опции обнаружения malloc.

output

По умолчанию отладочные опции malloc направляют вывод в stderr. Это не всегда бывает удобно. Опция `output` служит для указания другого канала вывода информации. Вывод можно направить в stderr, stdout или в любой файл.

Для того чтобы включить опцию `output`, укажите:

```
MALLOCDEBUG=output:<имя-файла>
```

continue

При обнаружении ошибки многие отладочные опции malloc вызывают функцию **abort()**. Это не всегда бывает удобно. Опция `continue` позволяет подсистеме **malloc** продолжить работу после обнаружения синхронной ошибки, вместо того чтобы прервать выполнение процесса. Сообщения об ошибках по-прежнему будут выводиться в соответствующие каналы.

Для того чтобы включить опцию `continue`, укажите:

```
MALLOCDEBUG=continue
```

Malloc debug fill

Malloc debug fill - это опция заполнения памяти, выделенной для отладки при помощи malloc(), применяемая вместе с пользовательским шаблоном.

В качестве шаблона должна быть указана строка с максимальной длиной 128 символов (например, export MALLOCDEBUG=fill:"abc" установит для выделяемой памяти шаблон "abc"). Если шаблон не указан, то данная опция игнорируется.

Для того чтобы ее включить, укажите:

```
MALLOCDEBUG=fill:шаблон
```

Шаблоном могут быть восьмеричные и шестнадцатеричные числа, указанные в виде строки. Например, шаблон "\101" - восьмеричное число, соответствующее символу 'A', а шаблон "\x41" - его шестнадцатеричный аналог.

Если указано недопустимое восьмеричное число, которое не может быть сохранено в 1 байте, то вместо него будет сохранено число \377 (максимальное допустимое восьмеричное число).

Понятия, связанные с данным:

"Выделение памяти в системе с помощью подсистемы malloc" на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Режим с несколькими кучами malloc

По умолчанию подсистема malloc использует одну кучу (пул свободной памяти).

Однако в ней предусмотрены средства для организации нескольких куч.

Организация нескольких куч в подсистеме **malloc** позволяет увеличить производительность многонитевых приложений в многопроцессорных системах. Если в подсистеме **malloc** применяется только одна куча, то запросы на выделение памяти, поступающие от нитей разных процессоров, обрабатываются по очереди. В этом случае подсистема **malloc** может в каждый момент времени обслуживать только одну нить, в результате чего значительно снижается производительность многопроцессорной системы.

В режиме нескольких куч подсистема **malloc** создает фиксированное количество куч. Как только процесс создает вторую нить (т.е. становится многонитевым), подсистема malloc начинает использовать несколько куч. Каждый запрос на выделение памяти обслуживается в одной из доступных куч. За счет этого подсистема **malloc** может распараллеливать запросы, поступающие от нескольких нитей, по крайней мере до тех пор, пока количество нитей не больше, чем число куч.

Если количество нитей, одновременно обращающихся к malloc, превысит количество куч, то лишние запросы будут обрабатываться по очереди. При условии, что такие ситуации возникают сравнительно редко, общая производительность подсистемы **malloc** значительно увеличится в случае, когда несколько нитей одновременно вызывают функцию **malloc** в многопроцессорной среде.

Включение режима нескольких куч подсистемы malloc

По умолчанию подсистема malloc работает с одной кучей. Для работы с несколькими кучами нужно задать переменную среды **MALLOCOPTIONS**. При задании перед запуском процесса опции `MALLOCOPTIONS=multiheap` поддержка нескольких куч будет использовать параметры по умолчанию. При таком значении переменной **MALLOCOPTIONS** подсистема malloc перейдет в режим нескольких куч со стандартными параметрами (32 кучи и быстрый алгоритм выбора кучи).

Параметры режима нескольких куч malloc

Параметры режима нескольких куч malloc перечислены ниже:

- multiheap:n
- considersize

Все эти опции подробно описаны ниже.

Они задаются в следующем формате:

```
MALLOCOPTIONS=[multiheap:n] | [considersize]
```

Можно указать одну из опций или две опции через запятую. Порядок опций не важен. Например:

```
MALLOCOPTIONS=multiheap:3,considersize
```

В данном случае будет применяться три кучи и медленный алгоритм выбора кучи, минимизирующий размер процесса.

Каждая опция конфигурации может быть указана в переменной **MALLOCOPTIONS** не более одного раза. Если какая-либо опция указана несколько раз, то будет применяться последнее из указанных значений.

Параметры режима нескольких куч malloc описаны ниже:

multiheap:n
: По умолчанию максимальное число куч равно 32. С помощью опции *multiheap:n* можно изменить этот максимум на любое число от 1 до 32, указав его в параметре *n*. Если *n* будет находиться вне допустимого диапазона, будет применяться значение по умолчанию (32). Включайте ровно столько куч, сколько их нужно процессам. Лишние кучи приведут к увеличению фрагментации и растрате ресурсов.

considersize
: По умолчанию при работе с несколькими кучами подсистема malloc направляет каждый запрос в следующую свободную кучу. Если будет указана опция *considersize*, подсистема malloc будет применять другой алгоритм выбора кучи, который выбирает ту свободную кучу, в которой достаточно памяти для выполнения запроса. Это позволяет сократить рабочий набор процесса за счет уменьшения количества вызовов функции **sbrk**. С другой стороны, в режиме *considersize* алгоритм выбора кучи выполняет дополнительные действия, поэтому он работает медленнее, чем стандартный алгоритм.

Если в кучах невозможно выделить память, то функция **malloc** возвращает значение NULL и код ошибки errno, равный ENOMEM. Если в текущей куче недостаточно памяти, то подсистема **malloc** попытается выделить память в другой свободной куче.

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Сегменты malloc

Наборы функции malloc представляют собой дополнительные расширения стандартной функции распределения памяти на основе наборов.

Эти наборы предназначены для повышения производительности приложений, создающих много запросов на выделение небольшого объема памяти. Если это расширение включено, то запросы на выделение блоков памяти, размер которых соответствует заданному диапазону, обрабатываются наборами функции malloc. Все остальные запросы обрабатываются обычным образом стандартной программой распределения памяти.

По умолчанию наборы malloc отключены. Для работы с несколькими кучами нужно задать переменную среды **MALLOCOPTIONS** до запуска процесса.

Размер и состав набора

Набор состоит из блока памяти, который делится на заданное число меньших блоков равного размера, каждый из которых может быть выделен отдельно. Каждому набору соответствует свой номер. Первый набор имеет номер 0, второй набор - 1, третий - 2, и т.д. Первый набор имеет наименьший размер, размер каждого последующего набора больше, чем размер предыдущего, и вычисляется по описанной ниже формуле. В куче может быть не более 128 наборов.

Размер блока каждого набора кратен базовому размеру набора. Базовый размер набора - это размер первого набора. Размер блоков второго набора в два раза больше, размер блоков третьего набора - в три раза больше, и т.д. Таким образом, размер блока конкретного набора вычисляется по формуле:

```
размер блока = (номер набора + 1) *
 базовый размер набора
```

Например, если базовый размер набора равен 16, то размер блока первого набора (набора 0) будет равен 16 байт, второго набора (набора 1) - 32 байта, третьего набора (набора 2) - 48 байт и т.д.

Для того чтобы адреса, возвращаемые функциями подсистемы malloc, были правильно выровнены для всех типов данных, базовый размер набора должен быть кратен 8 в 32-разрядной среде и 16 в 64-разрядной среде.

Размер набора вычисляется по следующей формуле:

```
размер набора = число блоков в наборе *
(значение malloc +
             ((номер набора + 1) *  базовый размер набора))
```

Приведенная выше формула позволяет вычислить фактический размер каждого набора. В этой формуле значение malloc описывает размер внутренней структуры malloc, необходимой для каждого блока набора. Размер внутренней структуры составляет 8 байт в 32-разрядных приложениях и 16 байт в 64-разрядных приложениях. Этот объем памяти не выделяется для пользовательских данных, однако включается в общий размер набора.

Число блоков в наборе, число наборов и базовый размер набора можно задать с помощью переменной среды **MALLOCOPTIONS**.

Выделение памяти из наборов

Блоки выделяются из одного из наборов, если включена функция наборов malloc, а запрос на выделение памяти соответствует диапазону объемов памяти для набора. В целях экономии память выделяется из наименьшего возможного набора.

Если при получении запроса все блоки набора уже выделены, функция malloc автоматически увеличит набор для обработки запроса. При увеличении набора к нему добавляется число блоков, равное первоначальному числу блоков. Это значение можно настроить с помощью переменной среды **MALLOCOPTIONS**.

Поддержка нескольких куч

Поддержка в функции malloc нескольких куч позволяет реализовать работу с несколькими кучами malloc для повышения производительности приложений с нитями в системах с несколькими процессорами. Функция наборов malloc поддерживает до 128 наборов в куче. Это позволяет подсистеме **malloc** поддерживать параллельное применение наборов malloc с несколькими кучами для увеличения производительности процессов, работающих в системах с несколькими процессорами, за счет применения наборов.

Включение поддержки наборов malloc

По умолчанию наборы malloc не применяются. Для их применения необходимо задать следующие переменные среды:

- **MALLOCTYPE**
- **MALLOCOPTIONS**

При использовании наборов malloc значение переменной среды **MALLOCTYPE** должно указывать на распределитель по умолчанию. При задании `MALLOCOPTIONS=buckets` перед запуском процесса для работы с наборами malloc будут применяться параметры по умолчанию. Для того чтобы указать пользовательские параметры для наборов malloc, задайте перед запуском процесса `MALLOCOPTIONS=buckets,options`, где *options* - список параметров конфигурации, разделенных запятыми.

Опции настройки наборов malloc

Переменная среды **MALLOCOPTIONS** позволяет включить поддержку наборов malloc со следующими предопределенными опциями настройки:

```
number_of_buckets:n
bucket_sizing_factor:n
blocks_per_bucket:n
bucket_statistics:[stdout|stderr|pathname]
no_mallinfo
```

Эти опции подробно обсуждаются в разделе **MALLOCOPTIONS**.

Значение переменной среды **MALLOCOPTIONS** задается в следующем формате:

```
MALLOCOPTIONS=[buckets,[ number_of_buckets:n | bucket_sizing_factor:n | blocks_per_bucket:n |
bucket_statistics:[stdout|stderr|pathname] | no_mallinfo],...]
```

В определении может быть указано несколько опций (в любом порядке), разделенных запятыми, например:

```
MALLOCOPTIONS=buckets,number_of_buckets:128,bucket_sizing_factor:8,bucket_statistics:stderr
MALLOCOPTIONS=buckets,bucket_statistics:stdout,blocks_per_bucket:512
```

Для разделения опций настройки применяются запятые. При использовании других разделителей (например, пробелов) опции конфигурации будут интерпретированы неправильно.

Каждая опция конфигурации может быть указана в переменной **MALLOCOPTIONS** не более одного раза. Если какая-либо опция указана несколько раз, то будет применяться последнее из указанных значений.

Если для опции настройки указано неверное значение, функция наборов malloc запишет предупреждение в стандартный вывод сообщений об ошибках и продолжит работу с применением значения по умолчанию.

Подсистема **malloc** обрабатывает параметры конфигурации наборов malloc только в том случае, если задан параметр `buckets`, как показано в следующем примере:

```
MALLOCOPTIONS=number_of_buckets:8,buckets,bucket_statistics:stderr
```

Параметры наборов malloc

number_of_buckets:*n*

Опция number_of_buckets:*n* задает число наборов в куче, где *n* - число наборов. Значение *n* относится ко всем кучам.

По умолчанию число наборов в куче равно 16. Минимальное значение - 1. Максимальное значение - 128.

bucket_sizing_factor:*n*

Опция bucket_sizing_factor:*n* задает базовый размер набора, где *n* - базовый размер набора в байтах.

Значение параметра bucket_sizing_factor должно быть кратно 8 в 32-разрядном приложении, и 16 в 64-разрядном приложении. По умолчанию применяется коэффициент размера набора 32 для 32-разрядных приложений и 64 для 64-разрядных приложений.

blocks_per_bucket:*n*

Опция blocks_per_bucket:*n* задает первоначальное число блоков в каждом наборе, где *n* - число

блоков. Это значение применяется ко всем наборам. Кроме того, значение *n* указывает, на сколько блоков нужно расширить область памяти, когда будет исчерпан запас блоков, выделенных первоначально.

По умолчанию для параметра blocks_per_bucket установлено значение 1024.

bucket_statistics:[stdout|stderr|*pathname***]**

Если указана опция bucket_statistics, то подсистема **malloc** выводит статистический отчет по наборам malloc по завершении каждого процесса, вызывающего подсистему malloc. Этот отчет содержит информацию о параметрах наборов и числе запросов на выделение памяти, обработанных для каждого набора. Если включена поддержка нескольких куч malloc, то число запросов на выделение памяти, указанное для каждого набора, представляет собой сумму всех запросов на выделение памяти, выполненных для этого набора из всех куч.

Статистический отчет по наборам направляется в одно из следующих устройств вывода, в соответствии со значением опции bucket_statistics.

- **stdout** - стандартный вывод
- **stderr** - стандартный вывод ошибок
- *полное-имя-файла* - пользовательский файл

Если указан пользовательский файл, то статистические данные добавляются в конец файла.

Для процесса, выходные данные которого являются входными для другого процесса, не следует применять стандартный вывод.

По умолчанию опция bucket_statistics отключена.

Примечание: Для первого набора всегда будет показан по крайней мере один запрос на выделение памяти, соответствующий функции **atexit**, которая выдает статистический отчет. Статистические данные для процессов с нитями будут содержать дополнительные запросы на выделение памяти для некоторых наборов из-за вызовов подсистемы malloc из библиотеки pthreads.

no_mallinfo

Если указать `MALLOCOPTIONS=no_mallinfo`, параметр **mallinfo** отключается, и информация о куче, управляемой подсистемой **malloc**, не вносится в протокол.

Опции настройки наборов malloc по умолчанию

Приведенная ниже таблица содержит значения опций настройки наборов malloc, применяемые по умолчанию.

Опция настройки	Значение по умолчанию (32-разрядное приложение)	Значение по умолчанию (64-разрядное приложение)
число наборов в куче	16	16
коэффициент размера набора	32 Б	64 Б
диапазон выделения памяти	1 - 512 Б (включительно)	1 - 1024 Б (включительно)
первоначальное число блоков в наборе	1024	1024
статистический отчет по наборам	отключена	отключена

Стандартная конфигурация функции наборов malloc увеличивает производительность приложений, создающих большое число запросов на выделение небольшого объема памяти. Однако изменение конфигурации наборов с помощью переменной **MALLOCOPTIONS** иногда позволяет повысить производительность. Для изменения заданных по умолчанию значений необходимо знать требования приложения к оперативной памяти и особенности ее использования в приложении. Для оптимизации настройки наборов malloc можно воспользоваться опцией bucket_statistics.

Ограничения

Из-за различных требований и особенностей использования памяти применение наборов malloc может не повысить производительность некоторых приложений. По этой причине не рекомендуется включать поддержку наборов malloc во всей системе. Для максимальной производительности поддержку наборов malloc следует включать и настраивать для каждого приложения в отдельности.

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Функция трассировки malloc

Функция трассировки malloc - это дополнительная функция подсистемы malloc, применяемая вместе с трассировщиком.

Она сохраняет информацию о трассировке функций **malloc**, **realloc** и **free**, которая может применяться при обнаружении неполадок и анализе производительности.

По умолчанию функция трассировки malloc выключена, но ее можно включить и настроить до запуска процесса с помощью переменной среды **MALLOCDEBUG**.

События, регистрируемые функцией трассировки malloc

Функция трассировки malloc применяет следующие идентификаторы точек трассировки:

- HKWD_LIBC_MALL_SUBSYSTEM
- HKWD_LIBC_MALL_INTERNAL

Если включена трассировка HKWD_LIBC_MALL_SUBSYSTEM, то в подсистеме трассировки сохраняются входные параметры и возвращаемые значения для всех вызовов функций **malloc**, **realloc** и **free**.

Когда для HKWD_LIBC_MALL_INTERNAL включена трассировка, и задействованы средства отладки обнаружения malloc, все ошибки внутренних структур данных подсистемы malloc будут заноситься в протокол трассировки.

Включение функции трассировки malloc

По умолчанию функция трассировки malloc выключена. Ее можно включить и настроить с помощью переменной среды **MALLOCDEBUG**. Для включения функции трассировки malloc задайте значение переменной среды **MALLOCDEBUG** с помощью следующей команды:

```
MALLOCDEBUG=trace
```

Для включения других функций отладки malloc присвойте переменной среды **MALLOCDEBUG** следующее значение:

```
MALLOCDEBUG=[trace, другая-опция]
```

Ограничения

Функция отладки трассировки malloc совместима со следующими стратегиями и опциями malloc:

- Стратегия выделения памяти по умолчанию.
- Стратегия выделения памяти 3.1
- Стратегия выделения памяти Watson.
- Стратегия выделения памяти Watson2.
- Сегменты malloc
- Режим с несколькими кучами malloc
- Кэш нитей malloc

- Malloc Disclaim Option

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Протокол malloc

Протокол malloc - это дополнительное расширение подсистемы **malloc**, позволяющее пользователю получать информацию об активных областях, выделенных вызывающим процессом. Эти данные могут применяться для определения причин возникновения неполадок и анализа производительности.

Данные из протокола malloc

Для каждой активной области в протокол malloc заносятся следующие данные:

- Адрес, переданный в вызывающий процесс.
- Размер выделяемой области.
- Куча, из которой была выделена область.
- Данные трассировки стека вызывающей функции. Глубину трассировки можно настроить.

Дополнительно можно настроить занесение в протокол следующей информации:

- ИД вызывающего процесса.
- ИД вызывающей нити.
- Порядковый номер выделения (с момента начала работы процесса).
- Точное время выделения.

При ведении протокола malloc каждая успешная операция выделения памяти требует дополнительных затрат для сохранения метаданных. Для 32-разрядных приложений эти затраты составляют 50-100 байт, 64-разрядные расходуют в два раза больше. Точное значение зависит от текущих параметров.

Информация протокола malloc может быть получена следующими способами:

- Командой DBX подсистемы malloc.
- С помощью опции отладки `report_allocations`.

Включение протокола malloc

По умолчанию протокол malloc не ведется. Для включения функции ведения протокола malloc с параметрами по умолчанию присвойте переменной среды **MALLOCDEBUG** следующее значение:

```
MALLOCDEBUG=log
```

Для включения функции ведения протокола malloc с пользовательскими параметрами присвойте переменной среды **MALLOCDEBUG** следующее значение:

```
MALLOCDEBUG=log:extended,stack_depth:6
```

Примечание: Ниже перечислены значения по умолчанию для параметров протокола malloc:

extended
: Значение по умолчанию - выключено. Этот параметр позволяет включить ведение протокола метаданных операций выделения памяти, упомянутых выше. Он влияет на количество дополнительных ресурсов, требуемых для каждой операции. Эта опция не имеет эффекта, если `MALLOCTYPE` равен `watson2`.

глубина-стека
: Задает глубину стека вызовов, содержимое которого сохраняется при выполнении каждой операции

выделения памяти. Он влияет на количество дополнительных ресурсов, требуемых для каждой операции. Значение по умолчанию - 4, максимальное значение - 64.

Ограничения

Применение протокола malloc может привести к снижению производительности всех программ из-за выполнения дополнительных операций записи данных в память. Кроме того, возрастет объем используемой памяти.

Функция отладки протокола malloc совместима со следующими стратегиями и опциями malloc:

- Стратегия выделения памяти по умолчанию.
- Стратегия выделения памяти Watson.
- Стратегия выделения памяти Watson2.
- Сегменты malloc
- Режим с несколькими кучами malloc
- Кэш нитей malloc
- Опция malloc Disclaim

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Опция malloc disclaim

Функция отказа от памяти malloc - дополнительное расширение подсистемы malloc, позволяющее пользователю автоматически отказываться от памяти, возвращаемой функцией **free**.

Эта функция необходима в случаях, когда процесс зарезервировал большой объем пространства подкачки, но фактически не использует эту память.

По умолчанию функция отказа от памяти malloc выключена. Ее можно включить и настроить до запуска процесса, присвоив соответствующее значение переменной среды **MALLOCOPTIONS**:

```
MALLOCOPTIONS=disclaim
```

Понятия, связанные с данным:

“Выделение памяти в системе с помощью подсистемы malloc” на стр. 622
Для приложений память выделяется с помощью подсистемы **malloc**.

Обнаружение malloc

Обнаружение malloc - это дополнительная функция подсистемы **malloc**, предназначенная для обнаружения ошибок и создания отчетов. В отчетах указывается информация об ошибках, возникших в среде malloc, а также сведения о рекомендуемых действиях.

Обнаружение malloc выполняет три задачи:

- Обнаружение ошибки.
- Вывод сообщения об ошибке в stderr.
- Вывод сообщения об ошибке с помощью указанной функции приложения.

Обнаружение ошибки.

Некоторые ошибки подсистемы **malloc** обнаружить довольно легко. Например, удаление указателя с неверным адресом в куче обнаруживается синхронно при исполнении кода функции free. Гораздо труднее обнаружить ошибки асинхронных событий, таких как повреждение кучи. Для синхронной работы с такими типами сбоев существует опция обнаружения malloc `check_arena`. При каждом вызове процедур подсистемы

malloc выполняется проверка внутренних структур данных. Если обнаружено повреждение, сообщение об этом передается приложению. Таким образом, возникает дополнительная точка отладки для быстрого нахождения причин таких скрытых сбоев. Опцию `check_arena` можно задать с помощью переменной среды **MALLOCDEBUG** следующим образом:

```
MALLOCDEBUG=checkarena
```

Вывод сообщения об ошибке в stderr

Обычно подсистема **malloc** сообщает об ошибках с помощью кода возврата и задает значение переменной errno. Опция `verbose` обнаружения malloc позволяет передать эти значения в стандартный поток вывода ошибок программы. Таким образом ошибки malloc становятся видимыми. Опцию verbose можно задать с помощью переменной среды **MALLOCDEBUG** следующим образом:

```
MALLOCDEBUG=verbose
```

|Вывод сообщения об ошибке с помощью указанной функции приложения

Обнаружение malloc позволяет пользователю задать функцию, вызываемую подсистемой **malloc** при обнаружении ошибки. Сначала будет вызвана соответствующая функция приложения, и после возврата из нее будет выполнен возврат из функции обнаружения malloc. Таким образом приложение получает возможность выполнить все отладочные операции, необходимые для продолжения работы программы. Для этого приложение должно задать адрес глобальной функции **malloc_err_function** равным адресу функции обработки ошибки самого приложения. Например:

```
extern void (*malloc_err_function)(int, ...)
malloc_err_function = &application_malloc_err_hdl
```

Ограничения

Функция отладки обнаружения malloc совместима со следующими стратегиями и опциями malloc:

- Стратегия выделения памяти по умолчанию.
- Сегменты malloc.
- Стратегия выделения памяти Watson.
- Режим с несколькими кучами malloc
- Malloc Threadcache
- Malloc Disclaim Option

Настройка кэша нитей и работа с ним

Кэш нитей malloc содержит пул свободной памяти, доступный нитям процесса и позволяющий снизить конкуренцию за выделение памяти в глобальной куче.

Кэш создает фрагменты свободной памяти для будущего использования нитями согласно тому, как эти нити использовали память в прошлом. Если запрос на выделение памяти можно выполнить с помощью кэша нитей, то соответствующий свободный фрагмент удаляется из кэша и передается вызывающей процедуре. В противном случае запрос на выделение памяти выполняется с помощью глобальной кучи.

Стратегия выделения памяти в кэше нитей|

В тот момент, когда нить в первый раз запрашивает фрагмент памяти размером менее 4096 байт, кэш нитей предварительно выделяет у себя несколько фрагментов памяти того же размера, забирая их у глобальной кучи. При этом в кэше также резервирует фрагмент памяти большего размера для обслуживания будущих запросов. Если нить освобождает область памяти, то она также сохраняется в кэше нитей для обслуживания будущих запросов. Однако если при освобождении памяти размер кэша нитей превысит определенный порог, то половина элементов кэша возвращается системному диспетчеру памяти. По сути кэш нитей

работает в режиме "пакетной обработки", группируя отдельные запросы на выделение или освобождение памяти для единовременного выполнения. Конкуренция за глобальные кучи уменьшается, что часто приводит к повышению быстродействия.

Включение кэша нитей malloc

По умолчанию кэш нитей включен при использовании стратегии выделения памяти Watson. Для его отключения нужно задать до запуска процесса переменную среды **MALLOCOPTIONS** следующим образом:

```
$ MALLOCOPTIONS=threadcache:off
```

При использовании стратегии выделения памяти по умолчанию кэш нитей по умолчанию выключен. Для его включения нужно до запуска процесса следующим образом задать переменную среды **MALLOCOPTIONS**:

```
$ MALLOCOPTIONS=threadcache
```

Написание реентерабельных программ и программ с поддержкой нитей

В процессах с одной нитью только один поток управления, и обеспечивать реентерабельность кода или поддержку нитей не требуется. В процессах с несколькими нитями одни и те же функции и ресурсы могут использоваться несколькими потоками управления одновременно.

поэтому для обеспечения целостности ресурсов код должен быть реентерабельным и предусматривать поддержку нитей.

Оба понятия - реентерабельность и поддержка нитей - связаны со способом работы функций с ресурсами. Однако это различные свойства: функция может обладать одним из них, обоими или ни одним.

В этом разделе содержится информация о написании реентерабельных программ и программ с поддержкой нитей. В нем не обсуждаются вопросы повышения эффективности нитей, т.е. оптимального распараллеливания потоков. Добиться такой эффективности можно только за счет удачного алгоритма. Существующие программы с одной нитью можно преобразовать в эффективные программы с несколькими нитями, однако для этого потребуется полностью переработать алгоритм и переписать программу заново.

Реентерабельность

Реентерабельная функция не может ни хранить статические данные в промежутках между вызовами, ни возвращать указатель на статические данные. Все данные передаются из вызывающей функции. Реентерабельная функция не может вызывать нереентерабельную функцию.

Реентерабельную функцию часто (но не всегда) можно определить по внешнему интерфейсу и по характеру применения. Например, функция **strtok** нереентерабельна, так как она хранит строку, разбиваемую на маркеры. Функция **ctime** также нереентерабельна, поскольку она возвращает указатель на статические данные, изменяемые при каждом вызове.

Поддержка нитей

Функция с поддержкой нитей обеспечивает защиту общих ресурсов от одновременного доступа путем установки блокировок. Определить функцию с поддержкой нитей по внешнему интерфейсу невозможно, так как поддержка нитей реализуется только на уровне алгоритма.

В языке C локальные переменные динамически помещаются в стек. Поэтому любая функция, не работающая со статическими данными и другими общими ресурсами, очевидно обеспечивает поддержку нитей, как это показано в следующем примере:

```
/* функция с поддержкой нитей */
int diff(int x, int y)
{
        int delta;

        delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
}
```

Применение глобальных данных не обеспечивает поддержку нитей. Глобальные данные необходимо обрабатывать в рамках какой-либо одной нити или инкапсулировать для сериализации доступа к ним. Нить может считывать код ошибки из другой нити. В AIX у каждой нити собственное значение **errno**.

Создание реентерабельных функций

В большинстве случаев для преобразования нереентерабельной функции в реентерабельную достаточно переделать только интерфейс. Нереентерабельные функции не могут применяться в нескольких нитях одновременно. Кроме того, в некоторых нереентерабельных функциях невозможно обеспечить поддержку нитей.

Возврат данных

Многие нереентерабельные функции возвращают указатели на статические данные. Избежать этого можно следующими способами:

- Возвращать динамические данные. В этом случае освобождение памяти выполняется в вызывающей функции. Преимущество такого подхода заключается в том, что не нужно переделывать интерфейс. Однако при этом не гарантируется совместимость с остальными программами: при вызове измененной функции из программы с одной нитью программа не освободит память.
- Использовать память, выделенную в вызывающей функции. Это рекомендуемый подход, хотя он и требует переработки интерфейса.

Например, функция **strtoupper**, преобразующая строку к верхнему регистру, может выглядеть так:

```
/* нереентерабельная функция */
char *strtoupper(char *string)
{
        static char buffer[MAX_STRING_SIZE];
        int index;

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

Эта функция нереентерабельная (и не обеспечивает поддержку нитей). Если выбран первый способ преобразования функции в реентерабельную, то ее исправленный код может выглядеть так:

```
/* реентерабельная функция (неудачный вариант) */
char *strtoupper(char *string)
{
        char *buffer;
        int index;

        /* необходима проверка наличия ошибок! */
        buffer = malloc(MAX_STRING_SIZE);

        for (index = 0; string[index]; index++)
```

```
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

Более удачный вариант требует переработки интерфейса. Вызывающая функция должна выделить память для строки ввода и строки вывода, как в следующем фрагменте программы:

```
/* реентерабельная функция (более удачный вариант) */
char *strtoupper_r(char *in_str, char *out_str)
{
        int index;

        for (index = 0; in_str[index]; index++)
        out_str[index] = toupper(in_str[index]);
        out_str[index] = 0

        return out_str;
}
```

Стандартные нереентерабельные библиотеки языка C были переработаны именно способом выделения памяти в вызывающей функции.

Хранение данных в промежутках между вызовами

Хранить данные в промежутках между вызовами нельзя, так как вызовы могут осуществляться из разных нитей. Если какие-либо данные (например, рабочий буфер или указатель) необходимо сохранить после возврата из функции, то их следует хранить в вызывающей функции.

Рассмотрим следующий пример. Пусть функция должна возвращать следующую строчную букву данной строки, причем сама строка указывается только при первом вызове функции, как в функции **strtok**. При достижении конца строки функция должна возвращать 0. Код такой функции может выглядеть так:

```
/* нереентерабельная функция */
char lowercase_c(char *string)
{
        static char *buffer;
        static int index;
        char c = 0;

        /* сохранение строки при первом вызове */
        if (string != NULL) {
                buffer = string;
                index = 0;
        }

        /* поиск строчной буквы */
        for (; c = buffer[index]; index++) {
                if (islower(c)) {
                        index++;
                        break;
                }
        }
        return c;
}
```

h

Эта функция нереентерабельна. Для того чтобы она стала реентерабельной, необходимо, чтобы статические данные, т.е. переменная **index**, хранились в вызывающей функции. Переработанный код функции может выглядеть так:

```
/* реентерабельная функция */
char reentrant_lowercase_c(char *string, int *p_index)
{
        char c = 0;

        /* инициализация не выполняется - она уже выполнена в вызывающей функции */

        /* поиск строчной буквы */
        for (; c = string[*p_index]; (*p_index)++) {
                if (islower(c)) {
                        (*p_index)++;
                        break;
                  }
        }
        return c;
}
```

Изменились и интерфейс функции, и способ ее применения. Вызывающая функция должна предоставлять строку при каждом вызове этой функции и инициализировать переменную index нулем до первого вызова, как в следующем фрагменте программы:

```
char *my_string;
char my_char;
int my_index;
...
my_index = 0;
while (my_char = reentrant_lowercase_c(my_string, &my_index)) {
        ...
}
```

Создание функций с поддержкой нитей

В программах с несколькими нитями все функции, вызываемые из нескольких нитей, должны обеспечивать поддержку нитей. Однако существуют способы вызова в таких программах функций без поддержки нитей. Нереентерабельные функции обычно не являются функциями с поддержкой нитей, но после переработки в реентерабельные часто становятся таковыми.

Блокировка общих ресурсов

Функции, работающие со статическими данными или с любыми другими общими ресурсами, например, файлами и терминалами, должны для обеспечения поддержки нитей сериализовать доступ к этим ресурсам с помощью блокировок. Например, следующая функция не обеспечивает поддержку нитей:

```
/* функция без поддержки нитей */
int increment_counter()
{
        static int counter = 0;

        counter++;
        return counter;
}
```

Для того чтобы обеспечить поддержку нескольких нитей, статическую переменную **counter** необходимо защитить с помощью статической блокировки, как показано в следующем примере:

```
/* псевдокод функции с поддержкой нитей */
int increment_counter();
{
        static int counter = 0;
        static lock_type counter_lock = LOCK_INITIALIZER;

        pthread_mutex_lock(counter_lock);
```

```
        counter++;
        pthread_mutex_unlock(counter_lock);
        return counter;
}
```

В приложении с несколькими нитями, работающем с библиотекой нитей, для сериализации доступа к общим ресурсам следует применять взаимные блокировки. При работе с независимыми библиотеками может потребоваться использовать их вне контекста нитей и потому устанавливать блокировки других типов.

Способы безопасного применения функций без поддержки нитей

Существует несколько специальных способов безопасного применения функций без поддержки нитей в программах, в которых эти функции вызываются из нескольких нитей. Эти способы могут пригодиться, в частности, при подключении библиотеки без поддержки нитей к программе с несколькими нитями - например, для тестирования, либо если версия этой библиотеки с поддержкой нитей пока не разработана. Реализация этих способов представляет собой достаточно сложную задачу, так как требуется осуществить сериализацию обработки данной функции или даже группы функций. Существуют следующие способы:

- Первый способ - глобальная блокировка всей библиотеки. Блокировка устанавливается при каждом обращении к библиотеке (т.е. при каждом вызове библиотечной функции или при каждом обращении к библиотечной глобальной переменной). Недостаток такого подхода заключается в возможном снижении производительности, так как в любой момент времени к каким бы то ни было средствам библиотеки сможет обращаться только одна нить. Описанный ниже способ можно применять только в случае, если обращения к библиотеке редки, или же временно.

  ```
  /* псевдокод! */

  lock(library_lock);
  library_call();
  unlock(library_lock);

  lock(library_lock);
  x = library_var;
  unlock(library_lock);
  ```

- Второй способ - блокировка каждого отдельного компонента библиотеки (т.е. каждой функции и каждой глобальной переменной) или группы компонентов. Этот вариант значительно более трудоемок, но не приводит к снижению производительности. Так как эти способы должны применяться только в приложениях, но не в библиотеках, для защиты библиотек можно применять взаимные блокировки.

  ```
  /* псевдокод! */

  lock(library_moduleA_lock);
  library_moduleA_call();
  unlock(library_moduleA_lock);

  lock(library_moduleB_lock);
  x = library_moduleB_var;
  unlock(library_moduleB_lock);
  ```

Реентерабельные библиотеки с поддержкой нитей

Реентерабельные библиотеки с поддержкой нитей применяются во многих параллельных (а также асинхронных) средах программирования, а не только при программировании нитей. Рекомендуется применять и создавать только функции, обладающие свойствами реентерабельности и поддержки нитей.

Работа с библиотеками

Часть библиотек, входящих в комплект поставки Базовой операционной системы AIX, обеспечивают поддержку нитей. В текущей версии AIX это следующие библиотеки:

- Стандартная библиотека C (**libc.a**)
- Библиотека, обеспечивающая совместимость с Berkeley (**libbsd.a**)

Некоторые стандартные функции C, например, **ctime** и **strtok**, нереентерабельны. Имена реентерабельных версий этих функций отличаются суффиксом **_r** (знак подчеркивания и буква *_r*).

При написании программ с несколькими нитями следует применять реентерабельные функции. Например, следующий фрагмент кода:

```
token[0] = strtok(string, separators);
i = 0;
do {
        i++;
        token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

в программе с несколькими нитями необходимо переработать так:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
        i++;
        token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

К библиотеке без поддержки нитей может обращаться только одна нить программы. Ответственность за соблюдение этого правила несет разработчик программы; нарушение этого правила может привести к непредсказуемым результатам и даже сбою программы.

Преобразование библиотек

В этом разделе описаны основные этапы преобразования уже существующей библиотеки в реентерабельную библиотеку с поддержкой нитей. Содержимое этого раздела относится только к библиотекам языка C.

- Выявление экспортируемых глобальных переменных. Эти переменные обычно определяются в файлах заголовка с помощью ключевого слова **export**. Экспортируемые глобальные переменные необходимо инкапсулировать, т.е. сделать их закрытыми (объявить их в исходном коде библиотеки с помощью ключевого слова **static**) и создать для них операции чтения и записи.
- Выявление статических переменных и других общих ресурсов. Эти переменные обычно определяются с помощью ключевого слова **static**. Установка защиты посредством блокировок для всех общих ресурсов. Количество блокировок влияет на производительность библиотеки. Для инициализации блокировок можно воспользоваться функцией однократной инициализации.
- Выявление нереентерабельных функций и преобразование их в реентерабельные. Дополнительная информация приведена в разделе Создание реентерабельных функций.
- Выявление функций без поддержки нитей и преобразование их в функции с поддержкой нитей. Дополнительная информация приведена в разделе Создание функций с поддержкой нитей.

Понятия, связанные с данным:

“Разовая инициализация” на стр. 455
Некоторые библиотеки на языке C используют динамическую инициализацию, когда глобальная инициализация библиотеки выполняется при первом вызове процедуры из этой библиотеки.

Информация, связанная с данной:

admin

cdc

delta

get

prs

sccsdiff

sccsfile

Создание пакетов программного обеспечения для установки

Этот раздел посвящен подготовке программного обеспечения к установке с помощью команды **installp**.

В этой главе приведена подробная информация о том, в каком формате разработчики должны поставлять свое программное обеспечение конечным пользователям. Вы узнаете о том, какие файлы должны входить в установочный пакет программного обеспечения.

Установочный пакет представляет собой файл в формате backup, в который входят собственно файлы программного продукта, управляющие файлы и, иногда, файлы для настройки процедуры установки. Для установки и обновления программных продуктов применяется команда **installp**.

Установочный пакет состоит из произвольного числа независимых логических компонентов, называемых *наборами файлов*. Все наборы файлов, входящие в пакет, должны относиться к одному и тому же продукту.

Обновлением или *пакетом обновлений* называется пакет, при установке которого вносятся изменения в уже установленный набор файлов.

Стандартной системой в этом разделе называется любая система, за исключением бездисковых систем.

Примечание: Если электронная документация по продукту написана в формате HTML, то при установке продукта ее можно зарегистрировать в AIX Information Center. Справочная система AIX Information Center предоставляет средства для навигации и поиска в документации по системе. Для получения информации по использованию Information Center выберите Справку **Information Center** в панели навигации. Сведения об установке и настройке Information Center для AIX приведены в книге *AIX 5L версии 5.3: Справочник и руководство по установке*.

Требования к процедуре установки

- Установка должна выполняться *без* участия пользователя. Если продукт нужно настраивать вручную, то настройку следует выполнять до или после установки.
- Все зависящие друг от друга наборы файлов и обновления должны устанавливаться в ходе одной операции.
- Процедура установки не должна перезагружать систему. Допускается временное прекращение работы отдельных компонентов системы, а полная перезагрузка должна выполняться после завершения установки.

Требования к управляющей информации

Управляющая информация пакета должна включать следующие сведения:

- Все требования устанавливаемых наборов файлов к прочим наборам файлов.
- Все требования устанавливаемых наборов файлов к размеру файловых систем.

Формат установочного пакета

Установочный пакет должен представлять собой файл в формате backup, который команда **installp** сможет восстановить во время установки. Этот файл может поставляться на ленте, дискетах или на компакт-дисках.

Требования к компоновке пакетов

Для поддержки бездисковых клиентов и клиентов без данных в установочном пакете должны быть разделены файлы, которые устанавливаются на каждом компьютере (*компонент root*), и файлы, которые могут использоваться несколькими компьютерами одновременно (*компонент usr*). Все файлы компонента usr должны устанавливаться в файловой системе **/usr** или **/opt**.

При установке компонента root содержимое файловой системы **/usr** изменяться не должно. Во время установки компонента root на бездисковых клиентах или клиентах без данных запись в файловую систему **/usr** будет запрещена. Машинно–зависимый (корневой) компонент должен включать в себя все файлы, не находящиеся в файловых системах **/usr** или **/opt**.

Пакеты в разделах рабочей схемы

Обратите внимание на следующие особенности отдельных программных продуктов, связанные с поддержкой разделов рабочей схемы (WPAR). Для успешного развертывания программного продукта в разделе WPAR пакет не должен пытаться записывать данные в файловые системы **/usr** или **/opt** в ходе обработки корневого компонента, поскольку WPAR монтирует эти файловые системы в режиме только для чтения. Кроме того, все операции настройки продукта в системе должны выполняться из корневого компонента пакета.

Если набор файлов не предназначен для установки в разделе рабочей схемы, то в файле **lpp_name** следует указать атрибут PRIVATE.

Если в случае установки в разделе рабочей схемы требуется другая конфигурация набора файлов, то сценарий создания пакета проверяет выбранное для установки расположение с помощью переменной `INUWPAR`.

Если в случае установки в WPAR требуется другая конфигурация набора файлов, то может потребоваться дополнительная настройка при создании WPAR из системной копии, поскольку изначально набор файлов не был установлен в WPAR. Владельцы наборов файлов могут создать программы в каталогах `/usr/lib/wpars/wparconvert.d/usr` и `/usr/lib/wpars/wparconvert.d/root`, позволяющие преобразовать компоненты *usr* и root для поддержки системной копии WPAR. Все исполняемые файлы в этих каталогах выполняются в алфавитном порядке (локаль C) при первом запуске системной копии WPAR.

Данные реестра программного обеспечения

Информация о программном продукте и его компонентах хранится в базе данных Реестра программного обеспечения (SWVPD). SWVPD содержит набор команд и классы объектов Администратора объектных данных (ODM), предназначенные для обслуживания информации о программном продукте. Команды SWVPD позволяют пользователю запросить (**lslpp**) и проверить (**lppchk**) информацию об установленных программных продуктах. Объектные классы ODM задают диапазон и формат этой информации.

С помощью Администратора объектных данных команда **installp** добавляет в базу данных SWVPD следующую информацию:

- Имя программного продукта (например, bos.adt)
- Версия программного продукта
- Выпуск программного продукта, который определяет, какие изменения были внесены во внешний программный интерфейс продукта
- Уровень модификации программного продукта, который определяет, какие изменения, не связанные с внешним программным интерфейсом, были внесены в продукт
- Уровень исправления программного продукта, который определяет небольшие обновления, которые будут добавлены в продукт следующего уровня модификации
- Имена, контрольные суммы и размеры файлов, входящих в программный продукт или его компонент
- Состояние программного продукта: доступен, устанавливается, установлен, фиксируется, зафиксирован, аннулируется или содержит ошибку
- Уровень обслуживания и информацию APAR
- Имя целевого каталога и программы установки для пакетов программного обеспечения, устанавливаемых без применения команды installp (если применимо).

Компоненты установочного пакета

Для поддержки среды клиент-сервер установочный пакет должен быть разделен на следующие компоненты:

usr
Содержит файлы, которые могут одновременно использоваться несколькими системами с совместимыми аппаратными платформами. В стандартной системе эти файлы хранятся в файловой системе **/usr** или **/opt**.

корневой
Содержит файлы, которые не могут использоваться в нескольких системах. На каждом клиенте устанавливается собственная копия этого компонента. Как правило, в компонент root входят файлы, связанные с конфигурацией конкретной копии продукта в конкретной системе. В стандартной системе файлы компонента root хранятся в корневой файловой системе (/). Компонент root набора файлов должен поставляться в том же установочном пакете, что и компонент usr. Если набор файлов содержит компонент root, то он должен содержать и компонент usr.

Краткое описание файловых систем

Ниже приведено краткое описание ряда файловых систем и каталогов. Оно поможет вам при разделении пакета на компоненты root, usr и share.

Некоторые каталоги, относящиеся к компоненту root:

/dev
Файлы устройств локальной системы

/etc
Файлы конфигурации (например, **hosts** и **passwd**)

/sbin
Системные утилиты, необходимые для загрузки системы

/var
Файлы данных и протоколы, относящиеся к локальной системе

Некоторые каталоги, относящиеся к компоненту usr:

/usr/bin
Команды и сценарии (обычные исполняемые файлы)

/usr/sbin
Команды администрирования системы

/usr/include
Файлы include

/usr/lib
Библиотеки, прочие команды и данные, зависящие от архитектуры конкретной системы

/opt
Библиотеки, прочие команды и сценарии, связанные с продуктами, отличными от операционной системы

Соглашения о присвоении имен пакетам и наборам файлов

При выборе имен для пакета программ и наборов файлов следуйте следующим рекомендациям:

- Имя пакета (*Пакет*) должно начинаться с названия продукта. Если пакет состоит из одного набора файлов, имя набора файлов может совпадать с именем пакета (*Пакет*). Имена всех пакетов должны быть уникальными.
- Имена наборов файлов должны быть заданы в следующем формате:

  ```
  ИмяПродукта.ИмяПакета.ИмяНабораФайлов.расширение
  ```

где:

- `ИмяПродукта` обозначает продукт или группу решений.
- `ИмяПакета` обозначает функциональную группу внутри продукта.
- `ИмяНабораФайлов` (необязательно) обозначает отдельный устанавливаемый функциональный набор файлов и библиотек.
- `Расширение` (необязательно) служит для более точного описания содержимого.

- Имя набора файлов должно содержать не менее 2 символов и начинаться с буквы.
- Имена наборов файлов должны состоять только из символов ASCII. Допустимыми символами являются строчные и прописные буквы, цифры, символы подчеркивания (_), знаки плюс и минус (+ -). Точка (.) используется в качестве разделителя в имени набора файлов.
- Имя набора файлов не может заканчиваться точкой или запятой.
- Длина имени набора файлов не должна превышать 144 байта.
- Имена всех наборов файлов в составе пакета должны быть различными.

Наборы файлов, правила присвоения расширений

Расширение	Описание набора файлов
.adt	Средства разработки приложений
.com	Общий код, необходимый для работы нескольких наборов файлов
.compat	Средства обеспечения совместимости, которые могут быть удалены в следующих версиях
.diag	Поддержка средств диагностики
.fnt	Шрифты
.help. *Язык*	Файлы справки CDE для указанного языка
.loc.language	Локаль
.msg. *Язык*	Файлы сообщений для указанного языка
.rte	Среда выполнения продукта
.ucode	Микрокод

Особые соглашения о присвоении имен пакетам с драйверами

Команда **cfgmgr** (администратор настройки) автоматически устанавливает драйверы обнаруженных в системе устройств, если эти драйверы записаны на установочном носителе. При этом имена драйверов должны быть заданы в следующем формате:

устройства.*ид-типа-шины.ид-карты.расширение*

где:

- *тип-шины* задает тип шины, к которой подключается карта (например, для PCI это будет **pci**)
- *карта* задает уникальный шестнадцатеричный идентификатор типа карты
- *Расширение* указывает часть драйвера (например, rte - это расширение для выполнения, a diag - для диагностики.)

Например, карте Ethernet для шины PCI в администраторе настройки соответствует идентификатор `1410bb02`. Соответствующему пакету драйверов должно быть присвоено имя **devices.pci.1410bb02**. Набор файлов среды выполнения, входящий в состав этого пакета, должен называться **devices.pci.1410bb02.rte**.

Особые соглашения о присвоении имен каталогам сообщений

В команде installp предусмотрена опция автоматической установки каталогов сообщений. Если эта опция указана в командной строке, то система попытается установить с носителя наборы файлов с каталогами сообщений для основного языка системы при условии, что их имена заданы в следующем формате:

продукт`.msg.`*язык.группа*

Необязательный суффикс *.группа* применяется в тех случаях, когда для разных *групп* наборов файлов в продукте предусмотрены разные наборы файлов с каталогами сообщений. Теоретически все каталоги сообщений продукта можно объединить в один набор файлов.

Например, продукт `Super_Widget` может состоять из групп наборов файлов `plastic` и `metal`. Все каталоги сообщений продукта `Super_Widget` на русском языке могут поставляться в одном наборе файлов `Super_Widget.msg.ru_RU`. Если для групп наборов файлов `plastic` и `metal` требуются разные каталоги сообщений, то они могут поставляться в двух наборах файлов - `Super_Widget.msg.en_US.plastic` и `Super_Widget.msg.en_US.metal`.

Примечание: Если имя набора файлов с сообщениями соответствует указанному формату, то для автоматической установки отдельно от основного продукта для такого набора файлов *обязательно* должно быть указано (с помощью атрибута **instreq**), что для его установки требуется другой набор файлов данного продукта.

Имена файлов

Имена файлов, поставляемых в составе установочных пакетов, не должны содержать запятых и двоеточий. Эти символы используются в качестве разделителей в управляющих файлах сценариев установки. Следует дополнительно отметить, что в именах файлов могут применяться символы, не входящие в набор ASCII. Длина полного пути к файлу не должен превышать 128 символов.

Идентификатор уровня набора файлов

Уровень набора файлов, который также называется *уровнем*, *v.r.m.f* или *VRMF*, задается в следующем виде:
версия.выпуск.модификация.уровень-исправления

где:

- *Версия* - это номер версии, содержащий 1 или 2 цифры.
- *Выпуск* - это номер выпуска, содержащий 1 или 2 цифры.
- *Модификация* - это номер модификации, содержащий от 1 до 4 цифр.
- *Уровень-исправления* - это номер уровня исправления, содержащий от 1 до 4 цифр.

Базовым уровнем набора файлов называется начальный уровень набора файлов, установленный в системе. Базовый уровень содержит все файлы набора, тогда как обновления обычно содержат только новые и исправленные файлы.

Рекомендуется присваивать всем наборам файлов в пакете один и тот же уровень, хотя в пакетах формата AIX 4.1 это не обязательно.

С течением времени уровень набора файлов может только возрастать. При проверке версий команда **installp** полагает, что чем выше уровень, тем новее набор файлов.

Самой старшей считается первая цифра уровня, самой младшей - последняя (то есть, уровень `5.2.0.0` выше уровня `4.3.0.0`).

Состав пакета программного обеспечения

В этом разделе описаны файлы, входящие в установочные пакеты. Каталоги, в которых находятся эти файлы, зависят от типа установочного пакета. Если в обновлении в качестве одного из компонентов пути указано *имя-пакета*, то оно заменяется на последовательность *имя-пакета*/*имя-набора-файлов*/*уровень-набора-файлов*.

В компонент usr установочного пакета входят следующие файлы управления установкой:

- **./lpp_name**: В этом файле содержится информация об устанавливаемом или обновляемом пакете. В целях повышения производительности рекомендуется помещать файл **lpp_name** в начало архивного установочного файла.
- **./usr/lpp/*Пакет*/liblpp.a**: В этом архивном файле содержатся управляющие файлы, применяемые для установки или обновления компонента usr пакета программного обеспечения.
- Архив, в котором содержатся все новые и обновленные файлы, относящиеся к компоненту usr устанавливаемого продукта. Файлы хранятся в этом архиве в виде дерева (с указанием абсолютного пути к каждому из них).

Если в установочном пакете есть компонент root, то в него должны входить следующие файлы:

- **./usr/lpp/*Пакет*/inst_root/liblpp.a**: В этом библиотечном файле содержатся управляющие файлы, применяемые для установки или обновления компонента root пакета программного обеспечения.
- Данный файл представляет собой архив, в который входят все новые и обновленные файлы, относящиеся к компоненту root устанавливаемого пакета. Если пакет предназначен для установки базового уровня продукта, то эти файлы должны быть помещены в архив с сохранением относительного пути от каталога **./usr/lpp/*Пакет*/inst_root**

Примеры пакетов программного обеспечения

Пакет `farm.apps` содержит набор файлов `farm.apps.hog 4.1.0.0`. Набор `farm.apps.hog 4.1.0.0` состоит из следующих файлов:

```
/usr/bin/raisehog (из компонента usr)
/usr/sbin/sellhog
 (из компонента usr)

/etc/hog
 (из компонента root)
```

Пакет `farm.apps` должен содержать, как минимум, следующие файлы:

```
./lpp_name
./usr/lpp/farm.apps/liblpp.a
./usr/lpp/farm.apps/inst_root/liblpp.a
./usr/bin/raisehog
./usr/sbin/sellhog
./usr/lpp/farm.apps/inst_root/etc/hog
```

Предположим, что обновление `farm.apps.hog 4.1.0.3` заменяет следующие файлы:

```
/usr/sbin/sellhog
/etc/hog
```

В этом случае установочный пакет обновления должен содержать следующие файлы:

```
./lpp_name
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/liblpp.a
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/inst_root/liblpp.a
./usr/sbin/sellhog
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/inst_root/etc/hog
```

Примечание: Файл из компонента root в установочном пакете находится в каталоге **inst_root**. Файлы, относящиеся к компоненту root пакета, зависящему от системы, должны находиться в каталоге **inst_root**. Эти файлы устанавливаются в корневой файловой системе каждой системы, на которой устанавливается пакет. Фактически они просто копируются в корневую файловую систему из каталога **inst_root**. Напомним, что компонент usr состоит из файлов, которые могут одновременно использоваться несколькими компьютерами.

Информационный файл lpp_name

Информационный файл **lpp_name** должен содержаться в каждом установочном пакете. Из файла **lpp_name** команда **installp** получает информацию о самом пакете и всех входящих в него наборах файлов. Пример файла **lpp_name** для пакета с обновлением приведен на следующем рисунке. Цифры и стрелки на этом рисунке соответствуют полям приведенной ниже таблицы.

Имя поля	Формат	Separator	Описание
1. Формат	Целочисленный	Пробел	Указывает версию программы **installp**, для которой предназначен данный пакет. Возможны следующие значения: • 1 - AIX 3.1 • 3 - AIX 3.2 • 4 - AIX 4.1 и более поздние
2. Платформа	Символ	Пробел	Указывает платформу, для которой предназначен пакет. Возможны следующие значения: • R - RISC • I - Intel • N - без учета
3. Тип пакета	Символ	Пробел	Указывает категорию (базовый набор файлов или обновление) и тип пакета. Возможны следующие значения: • I - Базовый набор файлов • S - Простое обновление • SR - Обязательное простое обновление • ML - Обновление уровня обслуживания
4. Имя пакета	Символ	Пробел	Имя пакета программного обеспечения (*Пакет*).
	{	Символ новой строки	Указывает начало описания конкретного набора файлов.
5. Имя набора файлов	Символ	Пробел	Полное имя набора файлов. Это значение указывается в начале заголовка набора файлов или обновления набора файлов.
6. Уровень	См. описание	Пробел	Уровень устанавливаемого набора файлов. Формат: *версия.выпуск.модификация.уровень-исправления* **Примечание:** Уровень можно указать следующим образом: **<**, **>** и **=**. Пример: `*prereq bos.rte v<5` или `*prereq bos.rte v=5 r=3`.
7. Том	Целочисленный	Пробел	Номер тома, на котором находится набор файлов (если он поставляется на многотомном носителе).
8. Bosboot	Символ	Пробел	Указывает, нужно ли выполнить bosboot после завершения установки. Возможны следующие значения: • N - bosboot не требуется • b - bosboot требуется
9. Состав	Символ	Пробел	Указывает, какие компоненты входят в набор файлов. Возможны следующие значения: • B - usr и root • U - только usr
10. Язык	Символ	Пробел	Должно быть установлено значение, отображаемое при выборе локали C. Обычно это en_US (для русского языка - ru_RU).
11. Описание	Символ	# или символ новой строки	Описание набора файлов. Его длина ограничена 60 символами.
12. Комментарии	Символ	Символ новой строки	Необязательные дополнительные комментарии.
	[	Символ новой строки	Указывает начало тела описания набора файлов.
13. Информация о зависимостях	Описан далее в этой главе	Символ новой строки	(Необязательно) Зависимости устанавливаемого набора файлов от других наборов файлов или от их обновлений. Подробное описание этого поля приведено вслед за этой таблицей.
	%	Символ новой строки	Разделяет поля зависимостей и размера.

Имя поля	Формат	Separator	Описание
14. Размер и сведения о лицензионном соглашении	Описано далее в этом разделе	Символ новой строки	Сведения о размере и лицензионных соглашениях для каждого каталога набора файлов. Подробные сведения об этом приведены в разделе Информация о размере и лицензионных соглашениях далее в этой главе.
	%	Символ новой строки	Разделяет поля размера и информации о лицензии.
	%	Символ новой строки	Разделяет поля заменяемого ПО и информации о лицензии.
15. Информация о заменяемом ПО	Описано далее в этом разделе	Символ новой строки	Информация о наборе файлов, заменяемом при установке
	%	Символ новой строки	Разделяет поля информации о лицензии и исправлении.
16. Информация об исправлениях	Описано далее в этом разделе	Символ новой строки	Информация о том, какие исправления содержатся в устанавливаемом обновлении. Подробные сведения об этом приведены в разделе Информация об исправлении далее в этой главе.
	]	Символ новой строки	Указывает конец тела описания набора файлов.
	}	Символ новой строки	Указывает конец описания набора файлов.

```
1 23    4
| ||    |                6         7 8 9  10       11
4 RSfarm.apps {  |       |         | | |   |        |
 5--> farm.apps.hog04.01.0000.0003 1 N U en_US Hog Utilities
12-->#...
[
13--> *ifreq bos.farming.rte (4.2.0.0) 4.2.0.15
%
14--> /usr/sbin 48
14--> /usr/lpp/farm.apps/farm.apps.hog/4.1.0.3 280
14--> /usr/lpp/farm.apps/farm.apps.hog/inst_root/4.1.0.3.96
14--> /usr/lpp/SAVESPACE 48
14--> /lpp/SAVESPACE 32
14--> /usr/lpp/bos.hos/farm.apps.hog/inst_root/4.1.0.3/ etc 32
%
%
15--> ranch.hog 4.1.0.0
%
16--> IX51366 Свиньи несут яйца.
16--> IX81360 У поросят слишком много ушей.
]
}
```

Раздел с информацией о зависимостях

В этом разделе содержится информация о том, от каких наборов файлов и обновлений зависит устанавливаемый набор файлов. Набор файлов или обновление устанавливается только в том случае, если выполнены все условия, указанные в этом разделе.

Перед началом установки команда **installp** сравнивает текущее состояние устанавливаемых наборов файлов с требованиями, перечисленными в файле **lpp_name**. Если в команде **installp** указан флаг **-g**, то все недостающие необходимые наборы файлов автоматически добавляются в список устанавливаемого программного обеспечения. Наборы файлов устанавливаются в том порядке, чтобы на момент начала установки очередного набора файлов были выполнены все необходимые условия. Непосредственно перед установкой каждого набора файлов команда **installp** повторно проверяет соблюдение указанных условий.

Это позволяет гарантировать, что если при установке какого-либо набора файлов возникнет ошибка, то зависящие от него наборы файлов не будут установлены.

В приведенном ниже описании различных типов необходимого программного обеспечения *требуемый-уровень* обозначает минимальный уровень набора файлов, который должен быть установлен в системе. Кроме случаев, когда это явно запрещено (см. раздел Информация о заменяемом программном обеспечении), вместо указанного уровня в системе может быть установлен набор файлов более высокого уровня. Например, если в условии указано, что требуется набор файлов `plum.tree 2.2.0.0`, то при наличии в системе набора файлов `plum.tree 3.1.0.0` это условие будет считаться выполненным.

Необходимое программное обеспечение

Для успешной установки набора файлов в системе должны быть установлены все необходимые наборы файлов заданного или более высокого уровня. Если необходимый набор файлов находится в списке устанавливаемого программного обеспечения, команда **installp** изменяет порядок установки таким образом, чтобы этот набор файлов был установлен раньше того набора, который от него зависит. При этом такие наборы не обязательно являются необходимыми.

Синтаксис

prereq`набор-файлов требуемый-уровень`

Альтернативный формат

`набор-файловтребуемый-уровень`

По умолчанию считается, что для установки обновления в системе должен был установлен базовый уровень этого же набора файлов. Если это условие не должно распространяться на поставляемое вами обновление, вам следует явно задать другое условие. По умолчанию считается, что *версия* и *выпуск* базового набора файлов совпадают с версией и уровнем обновления. Если *уровень-исправления* обновления равен `0`, то считается, что *модификация* и *уровень-исправления* базового набора файлов равны `0`. В противном случае считается, что *модификация* базового набора файлов совпадает с *модификацией* обновления, а *уровень-исправления* базового набора файлов равен `0`. Например, по умолчанию для установки обновления уровня 4.1.3.2 требуется, чтобы в системе был установлен набор файлов уровня 4.1.3.0. Для установки обновления уровня 4.1.3.0 требуется, чтобы в системе был установлен набор файлов уровня 4.1.0.0.

Сопутствующее программное обеспечение

Для правильной работы набора файлов в системе должны быть установлены все указанные сопутствующие наборы файлов. По окончании установки команда **installp** проверяет наличие сопутствующего программного обеспечения, и в случае его отсутствия выдает предупреждающее сообщение. Сопутствующим программным обеспечением для набора файлов могут быть и наборы файлов из того же пакета.

Синтаксис

coreq `набор-файловтребуемый-уровень`

Условная зависимость

Указывает, что если в системе установлен набор файлов уровня *установленный-уровень*, то он должен быть заменен на уровень *требуемый-уровень*. Такие условия обычно применяются для формирования зависимостей между обновлениями. Ниже приведен пример данного условия:

```
*ifreq plum.tree (1.1.0.0) 1.1.2.3
```

Синтаксис

ifreq`набор-файлов`
`[(`*установленный-уровень*`)]`
`требуемый-уровень`

Если набор файлов `plum.tree` вообще не установлен в системе, то он не будет установлен. Если установлен любой из следующих уровней набора файлов `plum.tree`, то он не будет заменен на уровень `1.1.2.3`:

`1.1.2.3`
Этот уровень совпадает с *требуемым-уровнем*.

`1.2.0.0`
Данный базовый уровень не соответствует базовому уровню обновления 1.1.2.3.

`1.1.3.0`
Этот уровень выше, чем *требуемый-уровень*.

Если установлен любой из следующих уровней набора файлов `plum.tree`, то он будет заменен на уровень `1.1.2.3`:

`1.1.0.0`
Данный уровень в точностью соответствует значению *установленный-уровень*.

`1.1.2.0`
Базовый уровень данного уровня совпадает со значением *установленный-уровень*, и при этом данный уровень ниже, чем *требуемый-уровень*.

Параметр (*установленный-уровень*) не обязателен. Если он не указан, то предполагается, что у значений *установленный-уровень* и *требуемый-уровень* совпадают *версия* и *выпуск*. Если *уровень-исправления* в значении *требуемый-уровень* равен `0`, то *модификация* и *уровень-исправления установленного-уровня* тоже равны `0`. В противном случае *модификация* в значении *установленный-уровень* равна *модификации* в значении *требуемый-уровень*, а значение *уровень-исправления* равно `0`. Пример: если *требуемый-уровень* равен `4.1.1.1` и не указан параметр *установленный-уровень*, то *установленный-уровень* считается равным `4.1.1.0`. Если *требуемый-уровень* равен `4.1.1.0` и не задан *установленный-уровень*, то предполагается, что *установленный-уровень* равен `4.1.0.0`.

Дополнительное программное обеспечение

Указанный набор файлов автоматически устанавливается только в том случае, если соответствующий набор файлов уже установлен в системе или находится в списке устанавливаемого программного обеспечения. Кроме того, дополнительный набор файлов устанавливается в том случае, если он явно указан в командной строке. Для обновления набора файлов дополнительное программное обеспечение задать нельзя. Обычно эти условия применяются для того, чтобы запретить установку наборов файлов с каталогами сообщений при отсутствии других компонентов пакета (нет смысла устанавливать каталоги сообщений, если не установлены соответствующие программы).

Синтаксис

`*instreq` *`набор-файловтребуемый-уровень`*

Групповое условие

Групповое условие считается выполненным, если выполнено более, чем минимум из входящих в него условий. Групповое условие может содержать условия необходимого, сопутствующего и обновляемого программного обеспечения, а также вложенные групповые условия. Значение *минимум* перед значением {*список-условий*} указывает, сколько условий из *списка-условий* должны быть выполнены для того, чтобы групповое условие считалось выполненным. Например, если задано `>2`, то для выполнения группового условия требуется, чтобы были выполнены хотя бы три условия из *RequisiteExpressionList*.

Синтаксис

`>`*`Число`* { *`RequisiteExpressionList`* }

Примеры информации о зависимостях

1. Следующий пример иллюстрирует применение сопутствующего программного обеспечения. Предположим, что набор файлов book.create 12.30.0.0 не может применяться без наборов файлов layout.text 1.1.0.0 и index.generate 2.3.0.0, поэтому для набора файлов book.create 12.30.0.0 задано следующее условие:

   ```
   *coreq layout.text 1.1.0.0
   *coreq index.generate 2.3.0.0
   ```

   Набор файлов index.generate 3.1.0.0 удовлетворяет условию index.generate, поскольку уровень 3.1.0.0 выше требуемого уровня 2.3.0.0.

2. В следующем примере применяются несколько условий различных типов. Для набора файлов new.fileset.rte 1.1.0.0 заданы следующие условия:

   ```
   *prereq database.rte 1.2.0.0
   *coreq spreadsheet.rte 1.3.1.0
   *ifreq wordprocessorA.rte (4.1.0.0) 4.1.1.1
   *ifreq wordprocessorB.rte 4.1.1.1
   ```

   На момент установки набора файлов new.fileset.rte в системе должен быть установлен набор файлов database.rte уровня 1.2.0.0 или выше. Если наборы файлов database.rte и new.fileset.rte устанавливаются за один вызов команды installp, то набор файлов database.rte будет установлен раньше набора файлов new.fileset.rte.

   Для работы набора файлов new.fileset.rte необходим набор файлов spreadsheet.rte уровня 1.3.1.0 или выше. Набор файлов spreadsheet.rte не обязательно устанавливать до набора файлов new.fileset.rte при условии, что оба набора файлов будут установлены за один вызов команды installp. Если набор файлов spreadsheet.rte нужного уровня не будет установлен на момент окончания работы команды installp, то будет выдано предупреждение о том, что отсутствует сопутствующее программное обеспечение.

   Если в системе установлен набор файлов wordprocessorA.rte уровня 4.1.0.0 (или он устанавливается вместе с набором файлов new.fileset.rte), то будет установлено обновление wordprocessorA.rte уровня 4.1.1.1 и выше.

   Если в системе установлен набор файлов wordprocessorB.rte уровня 4.1.0.0 (или он устанавливается вместе с набором файлов new.fileset.rte), то будет установлено обновление wordprocessorB.rte уровня 4.1.1.1 и выше.

3. В следующем примере проиллюстрировано применение условия на дополнительное программное обеспечение. Набор файлов Super.msg.in_IN.Widget уровня 2.1.0.0 дополняет следующий набор файлов:

   ```
   *instreq Super.Widget 2.1.0.0
   ```

   Набор файлов Super.msg.fr_FR.Widget не будет установлен автоматически, если не установлен набор файлов Super.Widget. Набор файлов Super.msg.fr_FR.Widget можно установить, если набор файловSuper.Widget не установлен, но явно указан в списке устанавливаемых наборов файлов.

4. Следующий пример иллюстрирует групповые условия. Для установки набора файлов требуется, чтобы был установлен хотя бы один из необходимых наборов файлов (могут быть установлены и оба набора). Если установлен набор файлов spreadsheet_1.rte, то он должен соответствовать уровню 1.2.0.0 или более высокому, либо должен быть установлен набор файлов spreadsheet_2.rte уровня 1.3.0.0 или выше.

   ```
   >0 {
   *prereq spreadsheet_1.rte 1.2.0.0
   *prereq spreadsheet_2.rte 1.3.0.0
   }
   ```

Размер и сведения о лицензионном соглашении

В этом разделе содержится информация о том, сколько дискового пространства требуется для установки набора файлов, а также сведения о лицензионном соглашении.

Информация о размере

Программа installp перед установкой набора файлов проверяет, достаточно ли свободной памяти в системе, и если требования из этого раздела не выполнены, набор файлов не устанавливается. Информация о размере приводится в следующем виде:

Каталог объем-постоянной-памяти [*объем-временной-памяти*]

Помимо этого, можно задать требования к объему пространства подкачки и дополнительной рабочей памяти, которая потребуется во время установки. Для этого в поле пути к файлу нужно указать опции **PAGESPACE** и **INSTWORK**.

Каталог
Полный путь к каталогу, в котором должен быть доступен указанный объем свободной памяти.

объем-постоянной-памяти

Объем дисковой памяти, требуемый для установки или обновления набора файлов (в блоках по 512 байт). Фактически это суммарный размер новых и обновленных файлов. Кроме того, в следующих случаях это поле играет особое значение:

Если в поле *Каталог* указано **PAGESPACE**, то значение *объем-постоянной-памяти* задает объем пространства подкачки (в блоках по 512 байт), требуемый для установки.

Если в поле *Каталог* указано значение **INSTWORK**, то значение *объем-постоянной-памяти* задает объем памяти (в блоках по 512 байт), необходимый для распаковки управляющих файлов, применяемых во время установки. Эти управляющие файлы должны содержаться в архиве **liblpp.a**.

объем-временной-памяти

Объем дополнительной памяти, которая потребуется в указанном каталоге на время установки или обновления набора файлов (в блоках по 512 байт). После завершения установки эта память будет освобождена. Значение *объем-временной-памяти* указывать не обязательно. Временная память может потребоваться, например, в таких случаях, когда при установке требуется скомпоновать выполняемый файл из объектного. Другим примером может служить операция помещения объектного файла в библиотеку. Для помещения файла в библиотеку команда **installp** создает копию библиотеки, затем помещает файл в копию библиотеки, а после этого заменяет исходную библиотеку на копию. Память, необходимая для копии библиотеки, потребуется только на время архивации файла, и ее следует учесть в объеме временной памяти.

Если в поле *Каталог* указано **INSTWORK**, то значение *объем-временной-памяти* задает размер сжатого файла **liblpp.a** в блоках по 512 байт.

Ниже приведен пример раздела с информацией о размере:

```
/usr/bin        30
/lib            40 20
PAGESPACE       10
INSTWORK        10  6
```

Поскольку структуру монтирования файловых систем в дереве каталогов предугадать достаточно сложно, рекомендуется указывать в этом разделе как можно более подробную информацию. Например, будет гораздо надежнее указать отдельные записи для каталогов **/usr/bin** и **/usr/lib**, чем одну запись для каталога **/usr**, потому что каталоги **/usr/bin** и **/usr/lib** могут находиться в различных файловых системах и при этом быть смонтированными в каталоге **/usr**. Лучше всего указывать подробную информацию о каждом каталоге, в который нужно поместить файлы.

Размер обновлений должен быть задан с учетом заменяемых файлов, старые версии которых будут перемещены в каталоги **save**. Эти файлы сохраняются на случай, если при установке новой версии возникнут ошибки и придется восстанавливать старую версию. Для того чтобы задать требования к объему памяти для сохранения старых файлов, укажите следующие значения:

/usr/lpp/SAVESPACE
Каталог **save** для файлов компонента usr. По умолчанию все файлы компонента usr сохраняются в каталоге **/usr/lpp/***пакет*/*набор-файлов*/*уровень***.save**.

/lpp/SAVESPACE
Каталог **save** для файлов компонента root. По умолчанию все файлы компонента root сохраняются в каталоге **/lpp/***пакет*/*набор-файлов*/*уровень***.save**.

Сведения о лицензионном соглашении

Новое дополнение к процессу установки AIX позволяет владельцу продукта потребовать от пользователя *принять* лицензионное соглашение до начала установки продукта. Обычно команда **installp** считывает файл *lpp_name* с любого образа. Если в поле размера файла *lpp_name* есть записи LAF или LAR, то **installp** вызывает команду **inulag**, которая показывает в окне текст лицензионного соглашения и фиксирует принятие пользователем этого соглашения. Если пользователь отказывается принять лицензионное соглашение, установка продукта прерывается.

Где должен находится файл лицензии

Владелец продукта может размещать файл лицензии по своему усмотрению. Однако настоятельно рекомендуется поместить его в каталог /usr/swlag/LANG. Рекомендуемое имя файла Лицензионного соглашения имеет вид: ИмяПродукта_ВерсияВыпуск.la . Однако использовать именно такое имя и расположение необязательно. Команда **installp** и такое размещение просто предлагают способ доставки информации потребителям. Все утилиты и необходимое наполнение должны предоставляться вместе с продуктом.

Требования к переводу файла лицензии

Если возникает необходимость перевода файла лицензионного соглашения на поддерживаемые языки, то рекомендуется поместить текст на каждом языке в отдельный файл. Таким образом, перевод может потребовать создания нескольких файлов.

Способ поставки файла лицензии

Файл лицензии может поставляться в составе основного продукта или в виде отдельного набора файлов. Во многих продуктах создается отдельный набор файлов, предназначенный только для файлов лицензий. Такой подход позволяет создать для сложного продукта с большим числом компонентов один файл и один набор файлов лицензии и поставлять этот набор файлов на всех носителях, необходимых для установки различных компонентов, а не включать файлы в состав каждого компонента. В настоящее время рекомендуется присваивать такому набору файлов имя lpp.license или lpp.loc.license. Большинство продуктов в настоящее время используют первое имя. Если вы хотите, чтобы набор файлов лицензии был невидим для пользователя при обычной установке, то присвойте ему имя lpp.loc.license, поскольку выбирать для установки набор файлов лицензии не требуется.

Способ упаковки файла лицензии

Сам файл никогда не указывается в списке *fileset.al* или *fileset.inventory* для набора файлов. Команда **installp** находит лицензию по записи в разделе размера в файле *ProductName*. Применяются следующие типы записей:

LAF

License **A**greement **F**ile (файл лицензионного соглашения) - указывает команде **installp**, что этот файл поставляется в составе данного набора файлов.

Файлы лицензионных соглашений обозначаются в записи раздела размера следующим образом:

LAF<язык>размер-файла-лицензии

Термин Значение

LAF **L**icense **A**greement **F**ile - файл лицензионного соглашения

<lang> Язык, на который переведен файл. Обычно это записи вида en_US, ru_RU, fr_FR, Ja_JP и

zh_TW. Если <язык> не указан, то считается, что файл лицензионного соглашения не переведен и поставляется в кодировке ASCII. Если файл лицензионного соглашения переведен, то обязательно должен быть указан <язык>. В противном случае невозможно будет связать файл с записью обязательных требований.

файл-лицензии
Полный путь к файлу лицензии в установочном образе и в системе. Рекомендуется применять путь вида /usr/swlag/ru_RU/ИмяПродукта_ВерсияВыпуск.la

раздел Фактический размер файла лицензии в блоках по 512 байт, позволяющий команде **installp** выделить достаточное количество места для размещения в системе файла лицензии.

LAR

License **A**greement **R**equisite (обязательное лицензионное соглашение) - указывает команде **installp**, что данный набор файлов требует обязательной установки указанного файла лицензионного соглашения. Это не то же самое, что обязательное программное обеспечение, поскольку лицензионное соглашение находится в файле, а не в наборе файлов. Файлы и наборы файлов имеют различный формат и предназначены для различных целей. Их всегда следует четко различать.

Термин Значение

LAR **L**icense **A**greement **R**equisite - обязательное лицензионное соглашение

req_license_file
Полный путь к файлу лицензии, необходимому для установки данного набора файлов. Обычно в этой записи вместо фактического названия языка указывается значение %L, позволяющее просматривать только файл на требуемом языке, а не все файлы на всех языках.

Обычно файлы поставляются в наборе файлов, содержащем все записи **LAF**. Другие наборы файлов продукта, требующие наличия этой лицензии, содержат только запись **LAR**. Набор файлов, поставляемый с записями **LAF**, содержит также перечисленные файлы в полностью заданном каталоге в образе BFF, но эти файлы не перечислены в файлах *fileset.al* и *fileset.inventory* набора файлов. Архитектура электронных лицензий *не* требует регистрации файлов в реестре программного обеспечения SWVPD. Команда **installp** выполняет следующие действия:

1. Находит обязательный файл.
2. Проверяет систему и убеждается в том, что условия соглашения приняты.
3. Если условия соглашения не приняты
   a. Находит набор файлов, в котором размещается файл
   b. Извлекает из образа BFF только один файл лицензионного соглашения
   c. Показывает содержимое файла пользователю.

Пример набора файлов LAF

Ниже приведен пример набора файлов с файлами лицензий:

```
iced.tea.loc.license 03.01.0000.0000 1 N U en_US IcedTea Лицензионная информация о рецепте чая
[
%
INSTWORK 16 160
LAF/usr/swlag/de_DE/iced.tea.la 24
LAF/usr/swlag/DE_DE/iced.tea.la 24
LAF/usr/swlag/en_US/iced.tea.la 24
LAF/usr/swlag/EN_US/iced.tea.la 24
LAF/usr/swlag/es_ES/iced.tea.la 24
LAF/usr/swlag/ES_ES/iced.tea.la 24
LAF/usr/swlag/fr_FR/iced.tea.la 24
LAF/usr/swlag/FR_FR/iced.tea.la 24
LAF/usr/swlag/it_IT/iced.tea.la 24
LAF/usr/swlag/IT_IT/iced.tea.la 24
```

```
LAF/usr/swlag/ja_JP/iced.tea.la 24
LAF/usr/swlag/JA_JP/iced.tea.la 32
LAF/usr/swlag/Ja_JP/iced.tea.la 24
LAF/usr/swlag/ko_KR/iced.tea.la 24
LAF/usr/swlag/KO_KR/iced.tea.la 24
LAF/usr/swlag/pt_BR/iced.tea.la 24
LAF/usr/swlag/PT_BR/iced.tea.la 24
LAF/usr/swlag/ru_RU/iced.tea.la 24
LAF/usr/swlag/RU_RU/iced.tea.la 48
LAF/usr/swlag/zh_CN/iced.tea.la 16
LAF/usr/swlag/zh_TW/iced.tea.la 16
LAF/usr/swlag/Zh_TW/iced.tea.la 16
LAF/usr/swlag/ZH_TW/iced.tea.la 24
%
%
%
]
```

Пример набора файлов LAR

Ниже приведен пример набора файлов с обязательными файлами лицензий:

```
iced.tea.server 03.01.0000.0010 1 N B en_US Набор рецептов чая
[
*prereq bos.net.tcp.client 5.1.0.10
*coreq iced.tea.tools 5.1.0.10
*coreq Java14.sdk 1.4.0.1
%
/usr/bin 624
/usr/lib/objrepos 24
/usr/include 16
/usr/include/sys 56
/usr/lpp/iced.tea 22
/usr/samples/iced.tea 8
/usr/samples/iced.tea/server 504
/usr/lpp/iced.tea/inst_root/etc/tea 8
/usr/iced.tea 8
/usr/lpp/iced.tea/inst_root/etc/tea/Top 8
INSTWORK 208 96
/lpp/iced.tea 104
/etc/tea 8
/etc/objrepos 8
/etc/tea/Top 8
/tmp 0 6
LAR/usr/swlag/%L/iced.tea.la 0
%
%
%
]
```

Информация о заменяемом программном обеспечении

В этом разделе указываются сведения о том, какие уровни набора файлов могут (или не могут) быть заменены набором файлов с данным уровнем. Это необязательная информация, которая может указываться только в базовых пакетах установки наборов файлов в формате AIX 4.1 и в пакетах обновления наборов файлов в формате AIX 3.2.

В некоторых случаях очередной уровень набора файлов может заменять не все предыдущие уровни. В такой ситуации в файле **lpp_name** должен быть задан максимальный заменяемый уровень. Команда **installp** не заменит набор файлов, если его уровень старше уровня, указанного в этом разделе.

Обновление считается заменяющим предыдущее обновление только в том случае, если в нем содержатся те же файлы, процедуры настройки и требования к среде установки, что и в предыдущем обновлении. Команда **installp** считает, что старое обновление набора файлов можно заменить на очередное обновление в том случае, если выполнены следующие условия:

- У старого и нового обновлений совпадают версия, выпуск и уровень модификации, у обоих обновлений уровень исправления не равен нулю, и для нового обновления (с более высоким уровнем исправления) не указано, что для его установки требуется обновление более высокого уровня, чем уровень старого обновления.
- У старого и нового обновлений совпадают версия и выпуск, и для нового обновления (с более высоким уровнем модификации) не указано, что для его установки требуется обновление более высокого уровня, чем уровень старого обновления.

Предположим, что обновление `farm.apps.hog 4.1.0.1` заменяет файл `/usr/sbin/sellhog`. В то же время, обновление `farm.apps.hog 4.1.0.3` заменяет файлы `/usr/sbin/sellhog` и `/etc/hog`. В свою очередь обновление `farm.apps.hog 4.1.1.2` заменяет файл `/usr/bin/raisehog`.

Обновление `farm.apps.hog 4.1.0.3` заменяет обновление `farm.apps.hog 4.1.0.1`, так как оно заменяет те же файлы и относится к той же модификации базового набора файлов (`farm.apps.hog 4.1.0.0`).

Обновление `farm.apps.hog 4.1.1.2` не заменяет ни обновление `farm.apps.hog 4.1.0.3`, ни обновление `farm.apps.hog 4.1.0.1`, так как оно содержит не все файлы, заменяемые этими обновлениями, и относится к другой модификации базового набора файлов (`farm.apps.hog 4.1.1.0`). Обновление `farm.apps.hog 4.1.1.0` заменяет обновления `farm.apps.hog 4.1.0.1` и `farm.apps.hog 4.1.0.3`.

Информация о заменяемом программном обеспечении для базового уровня набора файлов

В установочном пакете набора файлов в формате AIX 4.1 могут содержаться следующие операторы заменяемого программного обеспечения:

`Оператор границы`
Указывает минимальный уровень набора файлов, с которым совместим данный уровень набора файлов. Данный уровень набора файлов считается несовместимым со всеми более низкими уровнями. Эта запись применяется в тех случаях, когда по причинам несовместимости новым уровнем набора файлов нельзя заменять некоторые очень старые уровни.

`Оператор совместимости`
Указывает, что данный набор файлов может использоваться вместо другого набора файлов. Операторы совместимости применяются в случаях, когда требуется переименовать старый набор файлов или удалить его из системы. Каждый набор файлов может замещаться только одним набором файлов. Для каждого набора файлов можно задать только один оператор совместимости.

В файле **lpp_name** для каждого набора файлов должно содержаться не более одного оператора границы и не более одного оператора совместимости.

Оператор границы состоит из имени набора файлов и минимального уровня, с которым совместим данный уровень набора файлов. Операторы границы применяются только в тех редких случаях, когда очередной уровень набора файлов был изменен настолько сильно, что он уже не может использоваться вместо старых уровней этого же набора файлов. В установочных пакетах всех последующих версий этого набора файлов также должен быть указан оператор границы.

Например, если набор файлов `Bad.Idea` был существенно переработан в версии 6.5.6.0, то в установочных пакетах всех последующих уровней набора файлов `Bad.Idea` должен содержаться оператор границы для уровня `6.5.6.0`. Благодаря этому набор файлов `Bad.Idea 6.5.4.0` не будет заменяться никакими уровнями набора файлов `Bad.Idea` выше, чем `6.5.6.0`, потому что он будет считаться несовместимым с ними.

Оператор совместимости состоит из имени набора файлов (отличного от имени устанавливаемого набора файлов) и уровня этого набора файлов. Оператор совместимости указывает, что набор файлов данного и всех предыдущих уровней может быть заменен на устанавливаемый набор файлов. Операторы совместимости применяются в случаях, когда требуется заменить или переименовать устаревший набор файлов, все функции которого есть в новом наборе. Уровень, указанный в операторе совместимости, должен быть не ниже последнего поставлявшегося уровня заменяемого набора файлов.

В качестве примера предположим, что набор файлов `Year.Full 19.91.0.0` был разделен на несколько отдельных наборов файлов и больше не поставляется. Только для одного из этих наборов файлов, например, для `Winter 19.94.0.0`, должен быть указан оператор совместимости с `Year.Full 19.94.0.0`. Этот оператор позволяет установить набор файлов `Winter 19.94.0.0` вместо набора файлов `Year.Full` уровня `19.94.0.0` и ниже в тех случаях, когда набор файлов Year.Full требуется для работы каких-либо других наборов файлов.

Обработка информации о заменяемом программном обеспечении

В команде **installp** предусмотрены следующие возможности для установки наборов файлов и обновлений, заменяющих другие наборы файлов и обновления:

- Если на установочном носителе нет набора файлов или обновления, запрошенного пользователем, то команда installp может установить заменяющий набор файлов или исправление, если найдет его на установочном носителе.

  Предположим, что пользователь вызвал команду **installp** с флагом **-g** (автоматическая установка необходимого программного обеспечения) для набора файлов `farm.apps.hog 4.1.0.2`. Если на установочном носителе записан только набор файлов `farm.apps.hog 4.1.0.4`, команда **installp** установит `farm.apps.hog 4.1.0.4` как заменяющий набор файлов.
- Если в системе и на установочном носителе нет необходимого набора файлов или обновления, то вместо него может использоваться заменяющий набор файлов или обновление.
- Если в запросе на установку обновления указан флаг **-g**, то будет установлено записанное на носителе обновление максимального возможного уровня.

  Если в команде **installp** указан флаг **-g**, то все необходимые обновления (как явно указанные в командной строке, так и добавленные в список установки после проверки требований) будут заменены на заменяющие обновления максимально высокого уровня, найденные на установочном носителе. В связи с этим, если пользователю требуется установить конкретный уровень обновления (не обязательно самый высокий), в команде **installp** не следует указывать флаг **-g**.
- Если после проверки требований к установке выяснится, что необходимо записать два уровня обновления, причем один из них будет заменять другой и они оба записаны на установочном носителе, то команда **installp** установит только обновление более высокого уровня.

  Если в последнем случае пользователь хочет последовательно установить с носителя два обновления различных уровней, ему потребуется вызвать команду **installp** для каждого из этих уровней. Обратите внимание на то, что эта операция потеряет всякий смысл, если в командной строке вместе с флагом установки будет указан флаг фиксации (**-ac**). При фиксации второго обновления первое будет удалено из системы.

Информация об исправлениях

Информацию об исправлениях указывать не обязательно. В этом разделе содержится код исправления и описание исправленной ошибки длиной до 60 символов. Код исправления представляет собой уникальный идентификатор данного исправления длиной до 16 символов. Коды исправлений, начинающиеся с символов **ix**, **iy**, **IY** и **IX** зарезервированы для разработчиков операционной системы.

Обновлением уровня обслуживания называется обновление, существенно изменяющее функциональные возможности продукта. Примерами обновлений уровня обслуживания могут служить пакеты

профилактического обслуживания (PMP). Идентификатор уровня обслуживания начинается с имени программного продукта (не пакета), за которым идет одна точка (**.**) и идентифицирующий уровень, например `farm.4.1.1.0`.

Библиотечный файл управления установкой - liblpp.a

Файл **liblpp.a** - это архивный файл, содержащий файлы для управления установкой пакета. В системах более поздних версий, чем AIX 4.3, файл **liblpp.a** можно создать с помощью команды **ar** с флагом *-g*, чтобы гарантировать создание 32-разрядного архива. В этом разделе описано большинство стандартных файлов, входящих в архив **liblpp.a**.

В именах управляющих файлов, упоминаемых в этом разделе, присутствует компонент *набор-файлов*. *Набор-файлов* задает имя отдельного набора файлов, устанавливаемого в составе пакета программного обеспечения. Например, файл со списком установки обозначается как *набор-файлов***.al**. Это означает, что список установки для компонента `bos.net.tcp.client` продукта `bos.net` хранится в файле `bos.net.tcp.client.al`.

Имена всех файлов, входящих в архив **liblpp.a** и не описанных в этой главе, должны удовлетворять следующим условиям:

- Если файл используется при установке конкретного набора файлов, его имя должно начинаться с префикса *набор-файлов* .
- Если файл нужен для установки нескольких наборов файлов, входящих в один пакет, его имя должно начинаться с префикса **lpp.** .

Далеко не все файлы, описанные в этом разделе, обязательно должны присутствовать в файле liblpp.a. Многие файлы нужны только в том случае, если при установке требуются функции, предоставляемые этими файлами. По умолчанию файлы применяются как при установке базовых уровней, так и при установке обновлений.

Файлы данных, содержащиеся в файле liblpp.a

*набор-файлов***`.al`**
Список установки. В этом файле хранится список файлов, которые должны быть восстановлены для установки данного набора файлов. Каждый файл должен быть задан на отдельной строке с указанием абсолютного пути, например: `./usr/bin/pickle`. Список установки требуется для всех пакетов и наборов файлов, содержащих по крайней мере один файл.

*набор-файлов***`.cfginfo`**
Файл с особыми инструкциями. В этом файле содержится список ключевых слов, по одному в строке. Каждое ключевое слово задает особые параметры набора файлов или обновления. В настоящее время поддерживается только одно ключевое слово - **BOOT** - указывающее, что после завершения установки должно быть выдано сообщение о необходимости перезагрузки системы.

*набор-файлов***`.cfgfiles`**
Список файлов, настроенных пользователем, а также инструкций, которые должны применяться при установке нового или имеющегося уровня набора файлов. Перед тем как восстановить файлы, перечисленные в файле *набор-файлов***.al**, система сохраняет файлы, перечисленные в файле *набор-файлов***.cfgfiles**. Затем сохраненные файлы обрабатываются в соответствии с инструкциями, указанными в файле *набор-файлов***.cfgfiles**.

*Набор-файлов***`.copyright`**
Информация об авторских правах, связанная с набором файлов. Этот файл состоит из полного названия программного продукта и сведений об авторских правах.

*набор-файлов***`.err`**
Файл с шаблонами ошибок, передаваемый команде **errupdate** для добавления и удаления записей, хранящихся в архиве шаблонов ошибок. Этот файл обычно используется драйверами устройств. Для

обеспечения возможности очистки команда **errupdate** создает файл *набор-файлов***.undo.err**. Формат файла *набор-файлов***.err** приведен в описании команды **errupdate**. Этот файл может входить только в корневую часть набора файлов.

*Набор-файлов***.fixdata**
Файл формата настройки. В этом файле содержится информация о том, какие ошибки исправлены в данном наборе файлов или обновлении.

*набор-файлов***.inventory**
Файл реестра. Этот файл содержит информацию о файлах набора или обновления, которая заносится в реестр программного обеспечения. Это стандартный файл настройки, в котором указаны инструкции по обработке всех устанавливаемых и обновляемых файлов.

*набор-файлов***.namelist**
Список устаревших наборов файлов, в состав которых входили файлы, теперь включенные в состав устанавливаемого набора файлов. Этот файл применяется только при изменении логической структуры программных продуктов.

*Набор-файлов***.odmadd или** *Набор-файлов***.*.odmadd**
Разделы, добавляемые в базы данных ODM.

*набор-файлов***.rm_inv**
Файл с информацией, удаляемой из реестра. Этот файл применяется только при изменении логической структуры программных продуктов. Он обязательно должен поставляться в тех случаях, когда устанавливаемый набор файлов не является непосредственной заменой устаревшего набора файлов. Это стандартный файл настройки, в котором указаны имена файлов, относящихся к устаревшим наборам файлов и потому удаляемых из системы.

*набор-файлов***.size**
Этот файл содержит информацию о дисковом пространстве, необходимом для файлов данного набора в описанном выше формате.

*набор-файлов***.trc**
Файл с шаблоном отчета о трассировке. Команда **trcupdate** применяет этот файл для добавления, замены и удаления содержимого трассировочных отчетов в файле **/etc/trcfmt**. Для обеспечения возможности очистки команда **trcupdate** создает файл *набор-файлов***.undo.trc**. Файлы *набор-файлов***.trc** могут поставляться только для компонента root.

lpp.acf
Файл управления архивом, единый для всего пакета. Этот файл необходим только для добавления и удаления элементов архивных файлов, уже существующих в системе. В каждой строке файла управления архивом должны быть указаны имя элемента архива во временном каталоге (причем это имя должно быть указано в файле *набор-файлов***.al**) и имя архивного файла, к которому относится этот элемент. Оба имени должны быть указаны с абсолютным путем:

```
./usr/ccs/lib/libc/member.o ./usr/ccs/lib/libc.a
```

lpp.README
Файл readme. Содержит информацию, с которой пользователь должен ознакомиться до начала работы с программным обеспечением. Этот файл не обязателен и может также называться **README**, **lpp.doc**, **lpp.instr** или **lpp.lps**.

productid
Идентификационный файл продукта. Этот файл должен состоять из одной строки, в которой указаны имя продукта, идентификатор продукта (длиной не более 20 символов) и необязательный код продукта (длиной не более 10 символов).

Необязательные исполняемые файлы, содержащиеся в файле liblpp.a

Выполняемые файлы, описанные в этом разделе, вызываются в процессе установки пакета. Если не указано иное, файлы, имена которых заканчиваются на **_i**, вызываются при установке, а файлы, имена которых заканчиваются на **_u**, - при обновлении продукта. Все файлы, описанные в этом разделе, не обязательны и

могут быть как сценариями оболочки, так и выполняемыми объектными модулями. Все программы должны завершаться с кодом возврата 0, кроме случаев, когда при выполнении программы выясняется, что продолжение установки невозможно.

Набор-файлов.**config или** *Набор-файлов*.**config_u**
Изменяет конфигурацию при завершении стандартной процедуры установки или обновления продукта. Файл *набор-файлов*.**config** вызывается только при установке продукта.

Набор-файлов.**odmdel или** *Набор-файлов*.***.odmdel**
Обновляет информацию в базе данных ODM перед тем, как в нее будут добавлены новые записи для устанавливаемого или обновляемого набора файлов. Для одного набора файлов могут применяться несколько файлов **odmdel**.

набор-файлов.**pre_d**
Указывает, можно ли удалить данный набор файлов. Если набор файлов можно удалить, данная программа должна выдавать код возврата 0 (ноль). По умолчанию предполагается, что можно удалить любой набор файлов. Если файл удалить нельзя, программа должна выдать сообщение об ошибке с описанием причины, по которой нельзя удалить файл.

Набор-файлов.**pre_i или** *Набор-файлов*.**pre_u**
Вызываются перед восстановлением или сохранением файлов из списка установки, но после удаления предыдущей версии данного набора файлов.

набор-файлов.**pre_rej**
Вызываются перед операцией аннулирования или предварительного просмотра перед аннулированием набора файлов. С помощью этого сценария следует определять возможность аннулирования набора файлов. Этот сценарий не должен выполнять никаких операций, которые могут изменить что-либо в системе. Если сценарий завершится с ненулевым кодом выхода, операция аннулирования не выполняется.

набор-файлов.**pre_rm**
Вызывается во время установки перед удалением файлов предыдущей версии набора файлов.

Набор-файлов.**post_i или** *Набор-файлов*.**post_u**
Вызываются после восстановления файлов из списка установки набора файлов или обновления.

набор-файлов.**unconfig***набор-файлов*.**unconfig_u**
Аннулируют результаты настройки, выполненной в ходе установки или обновления продукта при его удалении или отмене установки. Файл *набор-файлов*.**unconfig** вызывается только во время удаления.

набор-файлов.**unodmadd**
Удаляет из баз данных ODM записи, добавленные в ходе установки или обновления продукта.

Набор-файлов.**unpost_i или** *Набор-файлов*.**unpost_u**
Аннулирует результаты обработки, выполненной после восстановления файлов из установочного списка в ходе установки или обновления продукта, при его удалении или отмене установки.

Набор-файлов.**unpre_i или** *Набор-файлов*.**unpre_u**
Аннулирует результаты обработки, выполненной до восстановления файлов из установочного списка в ходе установки или обновления продукта, при его удалении или отмене установки.

Если какой-либо из этих программ требуется выполнить команду, которая может изменить конфигурацию устройств системы, перед выполнением этой команды данная программа должна проверить значение переменной среды INUCLIENTS. Если переменная INUCLIENTS определена, конфигурацию устройств системы изменять нельзя. В среде сетевой установки (NIM) программа **installp** используется в различных целях, и в некоторых случаях команда **installp** не должна выполнять часть обычно выполняемых операций. Переменная INUCLIENTS определяется в среде NIM в тех случаях, когда некоторые операции выполнять не следует.

Если стандартные процедуры не подходят для обработки вашего пакета, вы можете включить в файл **liblpp.a** дополнительные исполняемые файлы, перечисленные ниже. В этом случае программа **installp** будет

выполнять ваши файлы вместо соответствующих системных файлов. Ваши версии этих файлов должны выполнять все функции стандартных файлов; в противном случае результаты установки будут непредсказуемы. Рекомендуем вам создавать собственные файлы только на основе стандартных, и вообще пользоваться собственными файлами только в крайних случаях.

instal
Применяется вместо стандартного сценария установки **/usr/lib/instl/instal**. Команда **installp** вызывает этот файл при установке базового набора файлов из пакета.

lpp.cleanup
Применяется вместо стандартного сценария очистки **/usr/lib/instl/cleanup**. Команда **installp** вызывает этот файл при установке базового набора файлов или обновления, которое было частично установлено и должно быть удалено для перевода набора файлов в согласованное состояние.

lpp.deinstal
Применяется вместо стандартного сценария удаления набора файлов **/usr/lib/instl/deinstal**. Этот выполняемый файл должен находиться в каталоге **/usr/lpp/***имя-пакета*. Программа **installp** вызывает этот файл при удалении базового набора файлов.

lpp.reject
Применяется вместо стандартного сценария аннулирования **/usr/lib/instl/reject**. Команда **installp** вызывает этот файл при аннулировании обновления набора файлов. (Стандартный сценарий **/usr/lib/instl/reject** представляет собой ссылку на сценарий **/usr/lib/instl/cleanup**.)

update
Применяется вместо стандартного сценария обновления **/usr/lib/instl/update**. Команда **installp** вызывает этот файл при установке обновления набора файлов. (Стандартный сценарий **/usr/lib/instl/update** представляет собой ссылку на сценарий **/usr/lib/instl/instal**.)

Для обеспечения совместимости с командой **installp** ваши программы **instal** и **update** должны удовлетворять следующим требованиям:

- Обрабатывать все наборы файлов пакета. Программа может либо обрабатывать все наборы самостоятельно, либо вызывать отдельный исполняемый файл для каждого набора файлов.
- Сохранять старые версии заменяемых файлов с помощью команды **inusave**.
- Восстанавливать все файлы компонента usr с установочного носителя с помощью команды **inurest**.
- Копировать все необходимые файлы компонента root из каталога **/usr/lpp/***пакет***/inst_root** с помощью команды **inucp**.
- Создавать файл **$INUTEMPDIR/status** с информацией о результатах обработки всех устанавливаемых и обновляемых наборов файлов.
- Выдавать код возврата, соответствующий результатам установки. Если программа **instal** или **update** возвращает ненулевой код возврата, и при этом не существует файла **status**, то предполагается, что ни один набор файлов не обработан.

Необязательный исполняемый файл, содержащийся в файле набор-файлов.al

*набор-файлов***.unconfig_d**
Аннулирует результаты особой процедуры настройки, выполняемой при установке и обновлении данного набора файлов. Файл *набор-файлов***.unconfig_d** выполняется в том случае, если команда **installp** вызвана с флагом **-u**. Если этого файла нет и в командной строке указан флаг **-u**, то вызываются программы *набор-файлов***.unconfig**, *набор-файлов***.unpost_i** и *набор-файлов***.unpre_i**.

Подробное описание файлов управления установкойеФайл набор-файлов.cfgfiles

В файле *набор-файлов***.cfgfiles** перечислены файлы конфигурации, которые должны быть сохранены для перехода к новой версии набора файлов без потери пользовательских данных. Файл *набор-файлов***.cfgfiles** должен находиться в файле **liblpp.a**, относящемся к тому же компоненту (usr или root), что и файлы с пользовательскими данными.

Файл *набор-файлов***.cfgfiles** содержит список всех файлов, которые требуется сохранить. В каждой строке данного файла должен быть указан абсолютный путь к сохраняемому файлу, пробел и ключевое слово, указывающее способ обработки этого файла. Предусмотрены следующие ключевые слова:

preserve
После установки нового уровня набора файлов новая версия файла заменяется на старую, которая была предварительно сохранена в специальном каталоге. После этого сохраненная версия удаляется из каталога, в котором она была сохранена на время установки нового уровня набора файлов.

auto_merge
Предполагается, что при выполнении программы *набор-файлов***.post_i** специальные программы, предоставленные разработчиком, добавляют нужную информацию из новой версии файла в старую версию, хранящуюся в каталоге сохранения. После завершения работы программы *набор-файлов***.post_i** команда **installp** заменяет новую версию файла на старую версию из каталога сохранения, после чего удаляет из этого каталога старую версию.

hold_new
После установки нового уровня набора файлов новая версия файла заменяется на старую, которая была предварительно сохранена в специальном каталоге. Новая версия помещается в каталог сохранения вместо старой версии. Если пользователю потребуется новая версия файла, он сможет взять ее из этого каталога.

user_merge
Новая версия файла остается в системе, а старая сохраняется в специальном каталоге. Если пользователю по каким-либо причинам понадобится старая версия файла, он всегда сможет взять ее из этого каталога. Применять это ключевое слово не рекомендуется.

other
Это ключевое слово указывается в тех случаях, когда ни один из перечисленных выше вариантов не подходит. Команда **installp** ограничивается тем, что помещает старый файл в каталог сохранения. Предполагается, что вся дальнейшая обработка выполняется программами, поставляемыми разработчиком продукта. Ответственность за документирование операций, выполненных над файлом, возлагается на разработчика.

Все дополнительные операции над файлами конфигурации, которые нельзя выполнить стандартными средствами, должны выполняться программой *набор-файлов*.**post_i**.

Файлы конфигурации, перечисленные в файле *набор-файлов***.cfgfiles**, сохраняются в том же каталоге сохранения конфигурации, что и сам файл *набор-файлов***.cfgfiles**. Имя каталога сохранения хранится в переменной среды **MIGSAVE**. Для разных компонентов (usr, share и root) применяются различные каталоги сохранения. Они перечислены ниже:

/usr/lpp/save.config
Компонент usr

/lpp/save.config
Компонент root

Если набор сохраняемых файлов зависит от установленного уровня набора файлов, то в файле *набор-файлов***.cfgfiles** должны быть перечислены все файлы конфигурации, которые могут нуждаться в замене. Выбор конкретных файлов для сохранения должен осуществляться программой *набор-файлов***.post_i**.

Предположим, что в набор файлов **change.rte** входит один файл конфигурации. Этот файл относится к компоненту root, и поэтому он указан в файле **change.rte.cfgfiles** для компонента root:

```
/etc/change_user   user_merge
```

При переходе от старого набора файлов (**change.obj**) к набору файлов **change.rte** этот файл сохранить не удастся, потому что его формат был изменен. В то же время, этот файл может сохраняться при установке новых уровней **change.rte**. Для этого нужно написать сценарий **change.rte.post_i**, алгоритм работы которого

зависит от того, какой набор файлов был установлен в системе ранее. В этом случае изменения, внесенные пользователем в файл **/etc/change_user**, не будут потеряны при обновлении.

Сценарий **change.bar.post_i** может выглядеть следующим образом:

```
#! /bin/ksh
rc=0
grep -q change.rte $INSTALLED_LIST
if [$? = 0]
то
mv $MIGSAVE/etc/change_user/ /etc/change_user
rc=1
fi
exit $rc
```

Команда **installp** определяет и экспортирует переменную среды $INSTALLED_LIST. Описание файла конфигурации *набор-файлов***.installed_list** приведен в разделе Файлы управления установкой для продуктов с измененной структурой. В переменной *$MIGSAVE* хранится имя каталога, в котором сохраняются файлы конфигурации компонента root.

Если команде **installp** не удается найти какой-либо из файлов, указанных в файле *набор-файлов*.**cfgfiles**, то она не выдает никаких сообщений. Кроме того, команда **installp** не выдает сообщений о файлах, не найденных во время выполнения сценария *набор-файлов*.**post_i** (этот сценарий восстанавливает сохраненные файлы конфигурации после установки новой версии набора файлов). Если вы хотите, чтобы пользователь получал какие-либо сообщения в таких ситуациях, эти сообщения должны выдаваться вашими сценариями.

Рассмотрим пример применения файла *набор-файлов*. **cfgfiles**. Допустим, в наборе файлов `Product_X.option1` предусмотрено три файла конфигурации, относящихся к компоненту root. Файл `Product_X.option1.cfgfiles` включен в ту часть файла `liblpp.a`, которая относится к компоненту root. Содержимое этого файла приведено ниже:

```
./etc/cfg_leafpreserve
./etc/cfg_pudding hold_new
./etc/cfg_newtonpreserve
```

Файл набор-файлов.fixdata

набор-файлов`.fixdata`
: Стандартный файл настройки, в котором хранится информация о том, какие ошибки исправлены в данном обновлении или в новом базовом уровне набора файлов

Сведения из этого файла добавляются в базу данных исправлений. С помощью этой базы данных команда **instfix** определяет, какие исправления установлены в системе. Если в пакете есть файл *набор-файлов***.fixdata**, содержимое этого файла заносится в базу данных исправлений при установке пакета.

Для каждого исправления должен быть отведен отдельный раздел файла *набор-файлов*.**fixdata**. Разделы файла *набор-файлов***fixdata** задаются в следующем формате:

```
fix:
name = ключевое-слово
abstract = описание
type = {f | p}
filesets = имя-набора-файлов уровень-набора-файлов
[имя-набора-файлов уровень-набора-файлов ...]
[symptom = [симптом]]
```

- Длина значения *ключевое-слово* не должна превышать 16 символов.
- В поле *abstract* указывается краткое описание исправления длиной не более 60 символов.
- В поле **type** должно быть указано значение **f** (исправление) или **p** (обновление уровня обслуживания).
- В поле **filesets** должен содержаться список имен и уровней наборов файлов, разделенных символами новой строки.

- Значение *уровень-набора-файлов* указывает начальный уровень набора файлов, в котором полностью или частично исправлена ошибка.
- В необязательном поле *symptom* можно указать описание ошибки, устраненной данным исправлением. Длина поля *symptom* не ограничена.

Ниже приведен пример раздела файла *набор-файлов*.**fixdata** для ошибки MS21235. Эта ошибка исправлена в двух наборах файлов.

```
fix:
name = MS21235
abstract = 82-гигабайтовый дисковод не работает на Марсе
type = f
filesets = devices.mca.8d77.rte 12.3.6.13
devices.mca.8efc.rte 12.1.0.2
symptom = 82-гигабайтовые субатомные дискеты не работают в марсианской атмосфере.
```

Файл набор-файлов.inventory

набор-файлов.**inventory**
: В этом файле хранится дополнительная информация обо всех новых и измененных файлах в данном уровне набора файлов

sysck
: Команда, заносящая в базу данных программного обеспечения имя файла, имя продукта, тип, контрольную сумму, размер и информацию о символьных связях, указанные в файле *набор-файлов*.**inventory**.

Для каждого компонента, в котором есть новые или измененные файлы, должен поставляться отдельный файл *набор-файлов*.**inventory**. Если в установочном пакете есть компонент root, но в этом компоненте нет ни одного нового и измененного файла, то файл *набор-файла*.**inventory** для компонента root не требуется.

Примечание: В файле *набор-файлов*.**inventory** можно указывать информацию только о тех файлах, размер которых не превышает 2 ГБ. Если в вашем установочном пакете есть файлы размером более 2 ГБ, укажите их файле *набор-файлов*.**al**, но не задавайте их в файле *набор-файлов*.**inventory**. Команда **sysck** в настоящее время не поддерживает файлы размером более 2 ГБ; кроме того, во многих системах файловая система **/usr** по умолчанию не поддерживает файлы размером более 2 ГБ.

Файл inventory - это текстовый файл в стандартном формате файла настройки. Имена разделов этого файла представляют собой полные пути к устанавливаемым файлам. Имя раздела должно заканчиваться двоеточием (:), за которым следует символ новой строки. После имени раздела могут быть указаны атрибуты в формате *атрибут=значение*. Каждый атрибут должен быть указан в отдельной строке.

Разделы файла *набор-файлов*.**inventory** задаются в следующем формате:

```
inventory:
type     = type
class    = inventory,apply,C2_exclude,fileset
owner    = имя_владельца
group    = group_name
mode     = TCB | SUID | SGID,permissions
target   = полный_путь_имяфайла
link     = полный-путь-жесткой-связи [additional_hardlinks]
size     =<blank> | VOLATILE |size
checksum =<blank> | VOLATILE |"checksum"
```

Атрибут **Описание**

file_name Полный путь к файлу или связи относительно корневого каталога (./), после которого следует двоеточие (:) и символ новой строки. Максимальная длина полного пути равна 255 символам.

type Тип записи file_name. Допускаются следующие типы:

Ключевые слова
Значение

Файл Обычный файл

DIRECTORY
Каталог

SYMLINK
Полный путь к символьной связи

class Указывает, как следует обращаться к имени **file_name** во время установки. В этом поле должно быть указано по крайней мере два следующих ключевых слова:

Тип **Значение**

inventory
Указывает, что файл должен остаться после завершения установки. Файл добавляется в базу данных реестра программного обеспечения SWVPD. Эту возможность не следует применять вместе с указанием типа файла **A** в *fileset.il*.

apply Указывает, что файл необходимо восстановить с установочного носителя. Поле *file_name* указывается в списке применения (*fileset.al*). Эту возможность не следует применять вместе с указанием типа файла **I** в *fileset.il*.

C2_exclude
Указывает, что этот файл не нужно обрабатывать в системах с уровнем защиты C2. Если указан данный флаг, то файл должен быть также указан в списке *fileset.tcb*.

owner Задает владельца, которому файл будет принадлежать после установки. Не указывайте в этом поле uid. В качестве значения атрибута должно быть указано имя длиной не более 8 символов.

group Указывает группу файла. Не указывайте в этом поле gid. В качестве значения атрибута должно быть указано имя группы длиной не более 8 символов.

mode Указывает режим доступа к файлу. Режим доступа должен быть задан в восьмеричном формате. Перед числом, определяющим режим доступа, могут быть указаны произвольные модификаторы из следующего списка. Если указано несколько модификаторов, они должны быть разделены запятыми.

Модификаторы
Значение

TCB Сокращение от "Trusted Computing Base -а защищенная компьютерная база". Если файл имеет **SUID** root или **SGID** system, то он должен быть помечен как **TCB**.

SUID Для файла установлен бит ИД пользователя. Для записи **DIRECTORY** это не играет никакой роли.

SGID Для файла установлен бит ИД группы. В случае записи **DIRECTORY** все файлы, создаваемые в данном каталоге, будут иметь ту же группу, что и каталог.

permissions
Шестнадцатеричное трехзначное число, например, 644.
Примечание: Для типа SYMLINK должен быть указан режим 777. Другие записи недопустимы.

связь Список жестких связей с файлом. Если существует несколько жестких связей, то они должны быть перечислены через запятую. Жесткие связи должны находиться в том же каталоге, что и исходный файл. Для записей типа **SYMLINK**, ключевое слово **link** недопустимо. Максимальная длина полного пути - 255 символов.

target Допустим только для **type=SYMLINK**. Это полный путь к целевому файлу связи. Если ссылка создана из каталога `/usr/bin` в `/bin`, то в качестве имени файла будет указано `/bin`, а в качестве целевого имени - `/usr/bin`. Максимальная длина полного пути - 255 символов.

Атрибут	Описание
раздел	**Модификаторы** Значение *blank* Если это поле пустое, то размер файла определяется во время установки. Недостаток данного способа заключается в том, что в случае повреждения файла во время установки пользователь не получит соответствующего предупреждения. **VOLATILE** Если предполагается, что размер файла может изменяться, этому атрибуту должно быть присвоено значение **VOLATILE**. **раздел** Точный размер файла. **Примечание:** Не указывайте поле size для записей **DIRECTORY** и **SYMLINK**.
checksum	**Модификаторы** Значение *blank* Если это поле пустое, то при установке файла в базу данных реестра программного обеспечения SWVPD заносится значение, возвращаемое командой **sum -r**. **VOLATILE** Указывает размер файла в блоках. Если предполагается, что размер файла может изменяться, этому атрибуту должно быть присвоено значение **VOLATILE**. **checksum** Точная контрольная сумма, возвращаемая командой **sum -r**. Значение должно быть заключено в двойные кавычки. **Примечание:** Не указывайте поле checksum для записей **DIRECTORY** и **SYMLINK**.
набор-файлов	Указывает набор файлов, к котором относится файл.

Примечание: Если указанные символьные и жесткие связи не существуют, то команда **sysck** создаст их во время установки. Символьные связи компонента root должны быть указаны в файле *набор-файлов***.inventory**, относящемся к компоненту root.

Пример fileset.inventory

Следующий пример набора файлов fileset.inventory демонстрирует применение обозначения типа **type**.

```
/usr/bin:
        owner = bin
        group = bin
        mode = 755
        type = directory
        class = apply,inventory,bos.rte


/usr/bin/tcbck:
        owner = root
        group = security
        mode = TCB,SUID,550
        type = file
        class = apply,inventory,bos.rte.security
        size = 99770
        checksum = "17077     98 "


/usr/sbin/tsm:
        owner = root
        group = security
        mode = TCB,SUID,555
        links = /usr/sbin/getty,/usr/sbin/login
        class = apply,inventory,bos.rte,security
        size = 55086
        checksum = "57960     54 "
```

Файлы управления установкой для продуктов с измененной структурой

*набор-файлов***.installed_list**
Этот файл создается командой **installp** во время установки набора файлов, если данный набор файлов уже установлен в системе полностью или частично.

Для того чтобы узнать, установлен ли набор файлов *набор-файлов* или какие-либо из наборов файлов, перечисленных в файле *набор-файлов***.namelist** (если он есть), команда installp просматривает базу данных программного обеспечения. Если какие-либо компоненты уже установлены, то в файл *набор-файлов***.installed_list** заносятся имена и уровни установленных наборов файлов.

Если файла *набор-файлов***.installed_list** нет, он создается перед вызовом программ **rminstal** и **instal**. Файл *набор-файлов***.installed_list** может находиться в рабочем каталоге пакета, в каталоге *рабочий-каталог-пакета* или в одном из следующих каталогов:

/usr/lpp/
имя-пакета - для компонента usr

/lpp/ *имя-пакета* - для компонента root

В файле *набор-файлов***.installed_list** перечислены все установленные наборы файлов. Каждая запись содержит имя и обозначение уровня набора файлов.

Например, предположим, что при установке набора файлов `storm.rain1.2.0.0` команда **installp** обнаружила, что в системе уже установлен набор файлов `storm.rain1.1.0.0`. В этом случае команда **installp** создает файл *рабочий-каталог-пакета*`/storm.rain.installed_list` со следующей информацией:

```
storm.rain 1.1.0.0
```

Другой пример: допустим, в набор файлов `Baytown.com` входит файл `Baytown.com.namelist` следующего содержания:

```
Pelly.com
GooseCreek.rte
CedarBayou.stream
```

При установке набора файлов `Baytown.com2.3.0.0` команда **installp** обнаружила, что в системе уже установлен набор файлов `Pelly.com 1.2.3.0` и `CedarBayou.stream4.1.3.2`. В этом случае команда **installp** создаст файл *рабочий-каталог-пакета*`/Baytown.com.installed_list` со следующей информацией:

```
Pelly.obj 1.2.3.0
CedarBayou.stream 4.1.3.2
```

Файл набор-файлов.namelist

Атрибут	Описание
*набор-файлов***.namelist**	Этот файл необходим при изменении имени набора файлов, а также в том случае, если набор файлов содержит файлы, ранее входившие в другие наборы файлов. В этом файле перечислены все наборы файлов, в которых ранее содержались файлы из данного набора. Имя каждого набора файлов должно быть указано в отдельной строке.

Файл *набор-файлов*.**namelist** должен находиться в файле **liblpp.a**, относящемся к компоненту **usr** или **root**. Файл *набор-файлов***.namelist** может поставляться только с установочными пакетами базового уровня; его нельзя поставлять с обновлениями.

В начале установки команда **installp** просматривает реестр программного обеспечения (SWVPD) и пытается определить, установлены ли в системе какие-либо из наборов файлов, перечисленных в файле *набор-файлов*.**namelist**. При обнаружении хотя бы одного набора файлов команда **installp** создает файл *набор-файлов***.installed_list** и записывает в него имена и уровни найденных наборов файлов.

Рассмотрим пример применения файла *набор-файлов*.**namelist**. Пусть набор файлов small.business заменяет поставлявшийся ранее набор файлов family.business. В установочном пакете small.business содержится файл small.business.namelist, хранящийся в файле **liblpp.a** в разделе компонента usr. В данном случае в файле small.business.namelist должна быть указана следующая информация:

```
family.business
```

В качестве более сложного примера применения файла *набор-файлов*.**namelist** рассмотрим набор файлов, состоящий из двух частей: клиента и сервера. Наборы файлов LawPractice.client и LawPractice.server заменяют предыдущий набор файлов lawoffice.mgr. Помимо этого, в набор файлов LawPractice.server входит несколько файлов из устаревшего набора файлов BusinessOffice.mgr. В файле LawPractice.client.namelist, который находится в файле **liblpp.a** установочного пакета LawPractice, должна содержаться следующая информация:

```
lawoffice.mgr
```

В файле LawPractice.server.namelist из файла **liblpp.a** пакета LawPractice должна содержаться следующая информация:

```
lawoffice.mgr
BusinessOffice.mgr
```

Если файл *набор-файлов*.**namelist** содержит только одну запись, а новый набор файлов не полностью заменяет набор файлов, указанный в файле *набор-файлов*.**namelist**, то в файл **liblpp.a** должен быть включен набор файлов *набор-файлов*.**rm_inv**. По содержимому файлов *набор-файлов*.**namelist** и *набор-файлов*.**rm_inv** программа установки определяет, какие наборы файлов полностью заменяются устанавливаемым набором файлов. Если файл *набор-файлов*.**namelist** состоит из одной строки, а файл *набор-файлов*.**rm_inv** отсутствует, то предполагается, что новый набор файлов полностью заменяет указанный старый набор файлов. В этом случае старый набор файлов полностью удаляется из системы при установке нового (заменяющего) набора файлов, даже если в старом наборе файлов были файлы, которых нет в новом наборе файлов.

В предыдущих примерах набор файлов small.business полностью заменяет набор файлов family.business, поэтому для него не требуется создавать файл small.business.rm_inv. Однако набор файлов LawPractice.client не полностью заменяет набор файлов lawoffice.mgr, поэтому для него обязательно должен быть создан файл LawPractice.client.rm_inv, даже если он будет пустым.

Пример 3:

Наборы файлов bagel.shop.rte и bread.shop.rte на протяжении долгого времени поставлялись отдельно друг от друга. Теперь решено поставлять bagel.shop.rte в составе bread.shop.rte. Для этого необходимо включить в файл bread.shop.rte.namelist следующую информацию:

```
bread.shop.rte
bagel.shop.rte
```

Кроме того, в набор необходимо добавить пустой файл bread.shop.rte.rm_inv, указывающий, что все файлы из набора bagel.shop.rte следует удалить из системы.

Файл набор-файлов.rm_inv

Атрибут	Описание
набор-файлов.**rm_inv**	В этом файле содержится список файлов, связей и каталогов, которые должны быть удалены из системы, если они установлены.

Этот файл применяется в том случае, когда по сравнению с предыдущим уровнем набора файлов изменилась его логическая структура, и ранее установленные файлы нельзя удалить на основании записей базы данных реестра, связанных с набором файлов.

Если вам требуется просто переименовать набор файлов, то файл *набор-файлов***.rm_inv** создавать не нужно. Файл *набор-файлов***.rm_inv** необходим только в тех случаях, когда новый набор файлов содержит подмножество файлов одного или нескольких устаревших наборов файлов. Если файл *набор-файлов***.namelist** есть в установочном пакете, и в нем указано несколько наборов файлов, то для удаления старых версий файлов из системы обязательно потребуется файл *набор-файлов***.rm_inv**.

Файл *набор-файлов***.rm_inv** - это текстовый файл в стандартном формате файла настройки. Имена разделов этого файла представляют собой полные пути к файлам или каталогам, которые должны быть удалены из системы (в случае, если они установлены). Имя раздела должно заканчиваться двоеточием (**:**), за которым следует символ новой строки. После имени раздела могут быть указаны атрибуты в формате *атрибут=значение*. Атрибуты применяются для определения жестких и символьных связей, которые также требуется удалить. Каждый атрибут должен быть указан в отдельной строке. Допустимы следующие атрибуты:

Атрибут	Описание
links	Список жестких связей с файлом. Связи должны быть перечислены через запятую с указанием абсолютного пути.
symlinks	Список символьных связей с файлом. Связи должны быть перечислены через запятую с указанием абсолютного пути.

Примечание: Связи должны быть указаны дважды - один раз в отдельном разделе, а один раз - в качестве атрибута файла, которому соответствует связь.

В качестве примера предположим, что набор файлов `U.S.S.R 19.91.0.0` содержал следующие файлы в каталоге **/usr/lib**: `moscow`, `leningrad`, `kiev`, `odessa` и `petrograd` (символьная связь с файлом `leningrad`). Разработчики решили разделить набор файлов `U.S.S.R19.91.0.0` на два набора файлов: `Ukraine.lib 19.94.0.0` и `Russia.lib 19.94.0.0`. Набор файлов `Ukraine.lib` состоит из файлов `kiev` и `odessa`. Набор файлов `Russia.lib` содержит файл `moscow`. Файл `leningrad` устарел и заменен файлом `st.petersburg` в наборе файлов `Russia.lib`.

В этом случае необходимо создать файл `Ukraine.lib.rm_inv`, поскольку набор файлов `Ukraine.lib` не полностью заменяет набор файлов `U.S.S.R`. Файл `Ukraine.lib.rm_inv` должен быть пустым, так как при установке набора файлов `Ukraine.lib` не нужно удалять файлы, относящиеся к устаревшему набору файлов `U.S.S.R`.

В этом случае необходимо создать файл `Russia.lib.rm_inv`, поскольку набор файлов `Russia.lib` не полностью заменяет набор файлов `U.S.S.R`. Поскольку файл `leningrad` заменен другим файлом в наборе `Russia.lib.rm_inv`, его следует указать в файле Russia.lib.rm_inv. В этом случае файл leningrad будет автоматически удален при установке набора файлов `Russia.lib`. Файл `Russia.lib.rm_inv` должен содержать следующую информацию:

```
/usr/lib/leningrad:
 symlinks = /usr/lib/petrograd
/usr/lib/petrograd:
```

Установочные файлы для дополнительных дисковых подсистем

Если для поставляемого устройства, подключаемого к SCSI или системной шине, требуются собственные драйвер и процедуры настройки, то изготовитель этого устройства должен поставлять специальные установочные файлы. Эти файлы должны быть записаны в формате backup на дискете, входящей в комплект поставки устройства, и им должны быть присвоены имена **./signature** и **./startup**. В файле signature должна содержаться строка **target**. Файл startup должен представлять собой сценарий, который разархивирует нужные файлы с дискеты с помощью команды restore, а затем выполняет все действия по настройке и подготовке устройства к работе.

Формат дистрибутивных носителей

Установочные пакеты программных продуктов могут поставляться на следующих видах носителей:

В следующих разделах приведена информация о том, в каком формате должны поставляться установочные пакеты на различных видах носителей.

Магнитная лента

Если вы хотите записать установочные образы нескольких продуктов на одну ленту или один набор лент, то для каждой магнитной ленты должны быть выполнены следующие условия:

- Файл 1 должен быть пустым. (Он может быть непустым только у загрузочных магнитных лент.)
- Файл 2 должен быть пустым. (Он может быть непустым только у загрузочных магнитных лент.)
- Файл 3 должен представлять собой таблицу содержимого с информацией о том, какие установочные образы записаны на данном наборе магнитных лент. Соответственно, на каждой магнитной ленте набора должна быть записана одна и та же полная таблица содержимого, отличающаяся только порядковыми номерами лент в наборе.
- Файлы с 4 по (*N*+3) содержат установочные образы продуктов 1 - *N* в формате backup.
- Каждый установочный образ должен целиком располагаться на одной ленте.
- После каждого файла на ленте должна быть записана метка конца файла.

компакт-диск

Компакт-диски с установочными образами нескольких продуктов должны соответствовать протоколу RRGP (Rock Ridge Group Protocol). Установочные пакеты должны находиться в каталоге, содержащем следующие данные:

- Установочные образы в формате backup.
- Таблицу содержимого в файле **.toc** с информацией обо всех установочных образах, записанных в данном каталоге.

Если вы хотите создать многотомный установочный носитель из нескольких компакт-дисков, то должен быть выполнен ряд дополнительных условий.

Многотомный установочный носитель из компакт-дисков должен удовлетворять следующим требованиям:

- На каждом диске должен существовать каталог **/installp/mvCD** со следующей информацией:
  1. Файл содержимого (.toc) с информацией обо всех установочных образах, записанных на всех томах. На всех томах в каталоге **/usr/sys/mvCD** должен быть записан один и тот же файл **.toc**.
  2. Текстовый файл с именем volume_id, в котором в первой строке указан номер компакт-диска в наборе.
  3. Символьная связь с именем **vol%*n***, где ***n*** - десятичный номер диска в наборе. Эта символьная связь должна указывать на каталог продуктов, записанных на данном компакт-диске. Обычно эта связь указывает на **../ppc**.
- Файл с таблицей содержимого (**.toc**) в каталоге **/installp/mvCD** должен соответствовать обычным требованиям к этому файлу. Единственное отличие файла **.toc** для многотомного носителя заключается в том, что расположение каждого установочного образа должно начинаться с каталога **vol% *n***, где ***n*** - номер тома, на котором записан этот установочный образ.

AIX 5.2 Пример:

Набор файлов A записан в файле **A.bff** на томе 1. Набор файлов B записан в файле **B.bff** на томе 2. В таблице содержимого в каталоге **/installp/mvCD** расположение наборов файлов A и B указано как **vol%1/A.bff** и **vol%2/B.bff**, соответственно. В таблице содержимого в каталоге **/installp/ppc** тома 1 расположение A указано как **A.bff**. В таблице содержимого в каталоге **/installp/ppc** тома 2 расположение B указано как **B.bff**.

Структура каталогов на компакт-дисках в AIX 5.1 и более поздних версиях допускает применение альтернативных программ установки, а также поддерживает установку на различных платформах.

Дискеты

Если вы хотите создать многотомный установочный носитель из дискет, на дискетах должны быть записаны следующие файлы:

- Таблица содержимого с описанием образов, записанных на многотомном носителе.
- Установочные образы всех продуктов.

При записи файлов на дискеты должны быть выполнены следующие правила:

- Данные должны записываться в виде потока; в блоке 0 каждой дискеты должен быть записан ИД тома. Данные, записанные в блоке 1 очередного тома, рассматриваются как логическое продолжение данных, записанных в последнем блоке предыдущего тома.
- Начало каждого файла должно приходиться на начало 512-байтового блока.
- Первым файлом набора дискет должна быть таблица содержимого. Дополните этот файл таким образом, чтобы его последний сектор был заполнен нулевыми байтами (`x'00'`). В конце таблицы содержимого должен быть хотя бы один нулевой байт - признак границы. Если таблица содержимого занимает целое число секторов, следующий сектор должен быть целиком заполнен нулевыми байтами.
- После таблицы содержимого должны быть последовательно записаны установочные образы всех продуктов. Последний сектор каждого образа должен дополняться нулевыми байтами. Если установочный образ занимает целое число секторов, то дополнительный нулевой сектор не требуется.
- В блоке 0 каждой дискеты должен быть записан ИД тома. ИД тома записывается в отдельном секторе в следующем формате:

Позиция	**Описание**
Байты 0:3	Сигнатура. Установочным носителям выделена сигнатура `3609823513=x'D7298918'`.
Байты 4:15	Дата и время. Это значение должно быть указано в формате *месяц-день-час-минута-секунда-год*. В поле *час* должно быть указано значение от 00 до 23. Все поля даты и времени состоят из 2 цифр. Поэтому для обозначения марта в поле *месяц* должно быть указано `03`, а не `3`, а для обозначения 1994 года в поле *год* должно быть указано `94`, а не `1994`.
Байты 16:19	Двоичный номер дискеты в данном наборе. Первой дискете присваивается номер `x'00000001'`.
Байты 20:511	Двоичные нули.

Файл с таблицей содержимого

Имя поля	**Формат**	**Separator**	**Описание**
1. Том	Символ	Пробел	Для магнитной ленты и дискет в этом поле указывается номер тома, на котором записаны данные. Для жестких дисков и компакт-дисков в этом поле должен быть указан `0`.
2. Дата и время	*ММДДччммссГГ*	Пробел	Для магнитной ленты и дискет в этом поле указывается время создания тома 1. Для жестких дисков и компакт-дисков в этом поле указывается время создания файла **.toc**. Более подробная информация приведена в разделе Формат даты и времени далее в этой главе.
3. Формат заголовка	Символ	Символ новой строки	Число, указывающее формат файла с таблицей содержимого. Допустимы следующие значения: • 1 -AIX версии 3.1 • 2 - версия 3.2 • 3 -AIX версии 4.1 и более поздних версий • B - лента **mksysb** (нельзя применять с командой **installp**)
4. Расположение установочного образа продукта	Символ	Пробел	Для магнитной ленты и дискет в этом поле указывается строка в формате: *ттт:бббббб:ппппппп*. Подробные сведения об этом приведены в разделе Расположение установочного образа на дискете и магнитной ленте далее в этой главе. Для жестких дисков и компакт-дисков в этом поле указывается имя файла. Имя файла указывается без пути к нему.
5. Информация о пакете	Формат файла **lpp_name**	Символ новой строки	Содержимое файла **lpp_name** из данного образа. Подробные сведения об этом приведены в разделе Файл с информацией о пакете lpp_name.

Примечание: Поля 4 и 5 задаются для каждого установочного образа, записанного на носителе.

Формат даты и времени

Дата и время указываются в следующем формате:

месяц-день-час-минута-секунда-год

В поле *час* должно быть указано значение от 00 до 23. Все поля даты и времени состоят из 2 цифр. Поэтому для обозначения марта в поле *месяц* должно быть указано `03`, а не `3`, а для обозначения 1994 года в поле *год* должно быть указано `94`, а не `1994`.

Формат поля расположения

Расположение установочного образа указывается в формате *ттт*:*ббббб*:*ppppppp*:

Магнитная лента

ттт - номер тома магнитной ленты.

ббббб - номер файла на магнитной ленте.

ppppppp
- размер файла в байтах.

Дискета

ттт номер тома дискеты.

ббббб номер блока дискеты, с которого начинается файл.

ppppppp
- размер файла в байтах (с учетом двоичных нулей, дополняющих файл до целого числа блоков).

Перемещаемая установка AIX

Хотя перемещаемая установка AIX поддерживается стандартными утилитами установки AIX (такими как **installp**, **instfix**, **lslpp** и **lppchk**), использование перемещения необходимо приложениям, устанавливаемым в рабочем разделе. Причина этого состоит в том, что стандартная конфигурация System WPAR не включает перезаписываемые файловые системы**/usr** или **/opt**. Поэтому, возможно, потребуется перенастроить установку приложения для установки в местах, отличных от обычного расположения **/usr** или **/opt**.

В дополнение к возможности устанавливать наборы файлов в стандартном месте установки (“/”), администратор может устанавливать перемещаемые пакеты в альтернативные места установки. Это позволяет администратору выполнять следующие действия:

- Устанавливать несколько установок одного и того же пакета installp и управлять ими в одном экземпляре операционной системы AIX
- Устанавливать разные версии одного и того же пакета installp и управлять ими в одном экземпляре операционной системы AIX
- Использовать стандартные средства трассировки installp (такие как **lppchk**, **lslpp**, **instfix** и **inulag**) для проверки данных установки всех перемещенных экземпляров установки и выдачи отчетов по ним
- Прикреплять и отделять предустановленные расположения ПО в данной системе (хостинг приложений).

Терминология

путь к корневому каталогу установки
Базовый каталог установки приложения. Стандартный путь установки **installp** - "/".

путь к корневому каталогу по умолчанию
Путь к корневому каталогу установки по умолчанию ("/").

перемещенный путь установки
Любой путь к корневому каталогу установки, не являющийся стандартным. Путь может быть любым, за исключением "/". Имя пути не должно превышать 512 символов.

перемещаемые приложения
Приложение, которое может быть установлено в нестандартный корневой каталог установки.

USIL (Пользовательский путь установки)
Заданный пользователем путь к перемещенному каталогу установки экземпляра.

USIL

Пользовательский путь установки или USIL - это трассируемый перемещенный путь к каталогу установки, созданной администратором. Это расположение трассируется системой и может быть использовано как альтернативный путь установки для пакетов с поддержкой перемещения. Множественные установки и/или версии программных пакетов можно устанавливать в отдельные пользовательские каталоги в одной системе. Также можно подключить или отключить существующий путь установки в системе.

Каждый экземпляр пользовательского пути установки содержит собственный набор реестра программного обеспечения (SWVPD) во всех трех каталогах installp:

- <*InstallRoot*>**/etc/objrepos**
- <*InstallRoot*>**/usr/lib/objrepos**
- <*InstallRoot*>**/usr/share/lib/objrepos**

Notes:

1. Текущие классы объекта реестра программного обеспечения включают следующее:
   - product
   - lpp
   - inventory
   - history
   - исправление
   - vendor
   - lag
2. Каждый экземпляр реестра программного обеспечения отображает стандартную структуру реестра в каталоге перемещения.

Команды управления USIL

/usr/sbin/mkusil

Создает или подключает экземпляр пользовательского пути установки.

mkusil -R *RelocatePath* **-c** *Comments* [**X***Fa*]

-a Подключает существующую установку как экземпляр пользовательского пути установки

-c Комментарии, включаемые в определение пользовательского пути установки (можно видеть с помощью команды **lsusil**)

-R Путь к созданному каталогу пользовательского пути установки. Каталог должен существовать.

-X Автоматически увеличивать размер используемого пространства при необходимости.

/usr/sbin/lsusil

Перечисляет существующие экземпляры пользовательского пути установки.

lsusil [**-R** *RelocatePath* | **"ALL"**]

-R Путь к существующему каталогу пользовательского пути установки.

/usr/sbin/rmusil

Удаляет все существующие экземпляры.

rmusil -R *RelocatePath*

-R Путь к существующему каталогу пользовательского пути установки.

Примечание: Команда **rmusil** удаляет только ссылки на пользовательский путь установки в реестре программного обеспечения. Файлы в каталоге не удаляются.

/usr/sbin/chusil

Изменяет атрибут экземпляра пользовательского пути установки.

chusil -R *RelocatePath* **-c** *NewComments* [**X**]

-c Включить новые комментарии в определение пользовательского пути установки (можно видеть с помощью команды **lsusil**)

-R Путь к существующему каталогу пользовательского пути установки.

-X Автоматически увеличивать размер используемого пространства при необходимости.

Утилиты перемещаемой установки

Для сохранения локализация неполадок исходный код все изменения пользовательского пути установки локализуются в отдельно компилируемый модуль. Утилиты перемещаемой установки включают следующие модули пользовательского уровня:

- **/usr/sbin/mkusil**
- **/usr/sbin/rmusil**
- **/usr/sbin/lsusil**
- **/usr/sbin/chusil**

Примечание:

1. Каждая утилита помечена флагом **-R** *RelocatePath*.
2. При работе с перемещаемыми пакетами **installp** необходимо использовать вышеперечисленные утилиты.

Требования к пакетам перемещаемых установок

Пакетов приложения должен поддерживать перенос установки. Рекомендуется применять следующие указания:

- Пакет перемещаемой установки не должен записывать объекты реестра вне пути установки.
- Пакет перемещаемой установки не должен записывать пользовательские данные вне пути установки.
- Пакет перемещаемой установки должен содержать расширенный атрибут **RELOCATABLE** для каждого перемещаемого набора файлов. Набор файлов - это минимально переместимый устанавливаемый модуль.
- Пакет перемещаемой установки не должен иметь дополнений, вне каталога установки. Он может иметь дополнительные наборы файлов, установленные в стандартном собственном каталоге установки.

Требования к выполнению перемещаемых приложений

Проект приложения должен поддерживать запуск из среды установки. Рекомендуется применять следующие указания:

- Приложение должно содержать метод определения места установки или функционировать без зависимости от него.
- Приложение должно обращаться к исполняемым компонентам с учетом места установки.
- Приложение должно обращаться к компонентам данных с учетом места установки или коллективно использовать данные других экземпляров приложения.
- Приложение не должно вносить постоянные изменения вне места установки.

Объект класса ODM USIL connector

Объектный класса администратора объектных данных коннектора пользовательского пути установки находится в файле **/etc/objrepos/usilc** и содержит данные, соединяющие стандартный реестр программного обеспечения с экземплярами пользовательского пути установки.

Ниже приведен пример записи этого объектного класса, содержащегося в swvpd.cre:

```
/*  Коннектор расположения пользовательской установки  */
/*  Connects the default install path to all relocated install paths.  */
class   usilc {
        vchar   path[1024];       /* Пользовательский путь установки */
        vchar   comments[2048];   /* Комментарии пользовательского пути установки  */
        long    flags;            /* Флаги пользовательского пути установки  */
        };
```

Перечисляет все пути установки с опцией -R "ALL" или -R "all"

Команды **lslpp** и **lppchk** могут выполнять операции перечисления в местах установки, если используется синтаксис **-R "ALL"**.

Операции подключения и отключения

Операция подключения позволяет пользователю интегрировать существующий отключенный пользовательский путь установки в реестр программного обеспечения.

Например, администратор может создать главный экземпляр пользовательского пути установки с различными перемещаемыми приложениями, установленными для хостинга. Затем администратор копирует этот экземпляр пользовательского пути установки (или NFS монтирует его) в различные системы и использует функцию подключения для его интеграции в реестр программного обеспечения. Операция отключения удаляет ссылки на экземпляр пользовательского пути установки.

лицензирование installp

Новый экземпляр пользовательского пути установки имеет пустое лицензионное соглашение (лицензионное соглашение объектного класса администратора объектных данных **installp**). Для любых наборов файлов или лицензионных программа необходима лицензия в соответствии с соглашениями **installp**. Лицензии не распространяются на экземпляры пользовательских установок.

|Защищенная компьютерная база (TCB)

Установка экземпляров USIL не поддерживается системами с активированным TCB.

Переносимые реквизиты

Новая семантика упаковки указывает расположение перемещаемых реквизитов. Упаковщик может указать расположение реквизитов - в каталоге по умолчанию или в пользовательском каталоге.

Ниже приведены примеры семантики:

- prereq_ r = prereq в пользовательском каталоге
- ifreq_r = ifreq в пользовательском каталоге
- coreq_r = coreq в пользовательском каталоге
- instreq_r = instreq в пользовательском каталоге

Данные определенные типы реквизитов являются стандартными: **prereq**, **ifreq**, **coreq** и **instreq**). Данные реквизиты находятся в стандартном каталоге.

Изменения TOC для переносимых пакетов

Ниже приведен пример нового раздела реквизитов в файле таблица содержания:

```
sscp.rte.1.0.0.5.U.PRIVATE.bff 4 R S sscp {
sscp.rte 01.00.0000.0005 1 N B En_US Sscp
[
*coreq bos.games 1.1.1.1  <-- стандартный реквизит в стандартном разделе
*prereq bos.rte 1.1.1.1   <-- стандартный реквизит в стандартном разделе
%
/usr/bin 20
/etc 20
INSTWORK 72 40
%
%
%
IY99999  1 APAR text here.
%
RELOCATABLE <-- тег атрибута, указывающий перемещаемый пакет
%
*prereq bos.rte 1.1.1.1   <-- стандартный реквизит в пользовательском разделе
*coreq_r bos.games 1.1.1.1 <-- стандартный реквизит в пользовательском разделе
]
}
```

Notes:

1. Если раздел переносимых реквизитов присутствует при переносимой установке, то он используется в качестве раздела реквизитов при установке.
2. Если во время установки в новом каталоге нет раздела перемещаемого реквизита, используется стандартный раздел реквизита. Таким образом все реквизиты используются по умолчанию.
3. Установка по умолчанию (не переносимая) не использует раздел переносимых реквизитов.

Алгоритм работы команды installp

Команда	Описание
Установка	При установке базового уровня набора файлов старая версия этого набора файлов, если она была установлена, удаляется из системы, и поэтому это действие равносильно *фиксации*. В дальнейшем пользователь может удалить этот набор файлов из системы. При установке обновления предыдущий уровень набора файлов сохраняется, чтобы при необходимости это обновление можно было удалить. Установленные обновления в дальнейшем можно зафиксировать или аннулировать.
Фиксация	Фиксацией установленного обновления называется удаление из системы предыдущего уровня набора файлов. При фиксации текущий уровень набора файлов не изменяется.

Команда	Описание
Аннулирование	Аннулированием установленного обновления называется удаление обновления из системы и восстановление предыдущего уровня набора файлов, который был сохранен при установке этого обновления. После этого резервная копия предыдущего набора файлов удаляется из системы. Аннулировать можно только обновления. Операция аннулирования во многом схожа с операцией **очистки**, при которой аннулируются результаты неудачной установки.
удаление	Удалением называется полное удаление из системы набора файлов и всех его обновлений , независимо от их текущего состояния (они могут быть установлены, зафиксированы или повреждены). Операция удаления возможна только для базовых уровней наборов файлов.

Разработчик может предоставить собственные сценарии установки, аннулирования и удаления своего продукта.

Повторная установка набора файлов поверх того же уровня не равносильна удалению этого набора файлов с последующей установкой. Сценарий повторной установки (см. **/usr/lib/instl/rminstal**) удаляет из системы файлы текущей установленной версии, но не выполняет сценарии **unconfig** и **unpre***. В связи с этим, создаваемые вами сценарии не должны исходить из предположения о том, что был выполнен сценарий **unconfig**. При принятии решений относительно того, была ли удалена информация о конфигурации, сценарий **.config** должен предварительно проверять среду.

Предположим, что сценарий настройки набора файлов **ras.berry.rte** добавляет строку в файл **crontab** пользователя root. Если вы заново установите набор файлов **ras.berry.rte**, то в файле **crontab** окажется две записи, поскольку при повторной установке не выполняется сценарий **unconfig** (который должен удалять строку из файла **crontab**). Сценарий настройки в подобной ситуации должен сначала удалять существующую запись, а затем добавлять собственную.

Операция установки

В этом разделе описаны действия, выполняемые командой **installp** при установке базового уровня или обновления набора файлов.

1. Восстановить информационный файл **lpp_name** для указанного пакета с заданного устройства.
2. Убедиться, что все запрошенные наборы файлов записаны на установочном носителе.
3. Убедиться, что уровень запрошенных наборов файлов допускает установку этих наборов в системе.
4. Восстановить управляющие файлы из библиотеки **liblpp.a** в *каталог пакета* (**/usr/lpp/***имя-пакета* для пакетов **usr** и **usr/root**. Управляющие файлы для компонента **root** пакета **usr/root** находятся в файле **/usr/lpp/***имя-пакета***/inst_root/liblpp.a**).
5. Проверить требования к объему свободной дисковой памяти.
6. Проверить, что все *необходимое программное обеспечение* (наборы файлов, определенный уровень которых должен быть установлен в системе перед началом установки данных наборов файлов) уже установлено или выбрано для установки.
7. Определить, нужно ли принять какие-либо лицензионные соглашения для продолжения установки.
8. Если устанавливается базовый уровень набора файлов, то просмотреть реестр программного обеспечения (SWVPD) и проверить, установлен ли в системе какой-нибудь уровень этого *набора файлов* или какие-либо из наборов файлов, перечисленных в файле *набор-файлов***.namelist**. Если *набор-файлов* уже установлен, то записать имя и уровень установленного набора файла в файл *рабочий-каталог/набор-файлов***.installed_list**.

   Если *набор-файлов* не установлен, но установлены какие-либо наборы файлов, перечисленные в файле *набор-файлов.***namelist**, то сохранить их имена и уровни в файле *рабочий-каталог/набор-файлов***.installed_list**. *Рабочий-каталог* обычно совпадает с каталогом, в котором устанавливается весь пакет за исключением компонентов **root**, которые устанавливаются в каталог **/lpp/***имя-пакета*.
9. Если устанавливается базовый уровень набора файлов, то вызвать сценарий **/usr/lib/instl/rminstal** для каждого из устанавливаемых наборов файлов.

Примечание: (Если не указано иное, то к этому моменту существующие файлы должны быть уже восстановлены из библиотеки управляющих файлов **liblpp.a**).

a. Если файл *набор-файлов***.pre_rm** существует, то вызвать *набор-файлов***.pre_rm** для подготовки к удалению текущего уровня *набора файлов*.

b. Если существует файл ***рабочий-каталог****набор-файлов***.installed_list**, то переместить существующие файлы, перечисленные в файле *набор-файлов*.cfgfiles, в каталог сохранения (путь к каталогу указан в переменной среды **MIGSAVE**)."

c. Если в системе уже установлен какой-либо уровень *набора-файлов*, то удалить его файлы из системы и сведения об этом *наборе-файлов* из SWPD (за исключением хронологии).

d. Если существует файл **рабочий-каталог**/*набор-файлов***.installed_list** и при этом существует файл *набор-файлов***.rm_inv** или файл *набор-файлов*.**namelist** содержит более одного набора файлов, либо в файле *набор-файлов***.namelist** указан только `bos.obj`, то необходимо выполнить следующие действия:

   1) Удалить из системы файлы, перечисленные в файле *набор-файлов***.rm_inv**, и удалить из SWVPD информацию об этих файлах.

   2) Удалить из системы файлы, перечисленные в файле *набор-файлов***.inventory**, и удалить из SWVPD информацию об этих файлах.

   3) Удалите из SWVPD информацию обо всех наборах файлов, перечисленных в файле **набор-файлов.namelist**.

e. Если существует файл **рабочий-каталог**/*набор-файлов*.**installed_list**, содержащий только один набор файлов, и список *набор-файлов***.namelist** содержит только один набор файлов, то необходимо удалить файлы этого набора файлов, а также удалить из SWVPD информацию об этих файлах (за исключением хронологических сведений).

f. Для каждого компонента пакета (либо только для **usr**, либо для **usr**, а затем для **root**)

   1) Присвоить нужные значения переменным среды INUTREE (*U* для **usr** и *M* для **root**) и INUTEMPDIR (имя ранее созданного временного рабочего каталога).

   2) Если в каталоге пакета существует управляющая программа **instal** (создавать ее не рекомендуется), то запустить **./instal**, в противном случае запустить сценарий по умолчанию **/usr/lib/instl/instal** . Если управляющая программа **instal** не существует в каталоге пакета, то задать значение в переменной среды **INUSAVEDIR**.

   3) Если в каталоге пакета существует управляющая программа **update** (ее создавать не рекомендуется), то запустить **./update**. Если управляющая программа **update** не существует в каталоге пакета, то запустить сценарий по умолчанию **/usr/lib/instl/update**.

   4) Если файл **status** был успешно создан сценарием **instal** или **update**, то с помощью файла **status** определить результат установки каждого набора файлов. Если файл **status** не был создан, то сделать вывод, что все запрошенные наборы файлов из пакета не были установлены.

   5) Если набор файлов был успешно установлен, то обновить реестр программного обеспечения (SWPD) и зарегистрировать требования соответствующего лицензионного соглашения. Если набор файлов не был установлен, то запустить команду **/usr/lib/instl/cleanup** или находящийся в каталоге пакета сценарий **lpp.cleanup** для очистки данных после неудачной установки.

Алгоритм работы стандартного сценария установки или обновления

Команда **installp** передает в качестве первого параметра сценария установки **instal** или **update** имя установочного устройства. Во втором параметре указывается полный путь к файлу со списком устанавливаемых или обновляемых наборов файлов. Ниже этот параметр обозначается **$FILESETLIST**. Стандартные сценарии **instal** и **update** отличаются друг от друга. текущий каталог - это каталог пакета. В каталоге **/tmp** создается временный каталог INUTEMPDIR для хранения рабочих файлов.

Стандартные сценарии **instal** и **update** выполняют следующие действия:

1. Выполнить следующие действия для каждого набора файлов из **$FILESETLIST**:

a. Если набор файлов представляет собой обновление, то выполнить программу ***набор-файлов*.pre_u** (pre_update) при условии, что она существует. Если набор файлов не является обновлением, то выполнить программу ***набор-файлов*.pre_i** (pre_installation) при условии, что она существует.

b. Создать список файлов для восстановления из пакета, добавив содержимое файла ***набор-файлов*.al** в новый файл **INUTEMPDIR/master.al**.

c. Если выполняется обновление и необходимо сохранить какие-либо файлы, и при этом существует архивный управляющий файл **lpp.acf**, то

   Сохранить обновляемые элементы архива библиотек.

d. Если операция была выполнена успешно, то добавить текущий набор файлов в список устанавливаемых наборов файлов в файле **$FILESETLIST.new**.

2. Если устанавливается обновление, и необходимо сохранить какие-либо файлы, то запустить сценарий **inusave** для сохранения текущих версий файлов.
3. При обработке компонента root запустить сценарий **inucp** для копирования файлов из списка установки компонента root. Если обрабатывается другой компонент, то запустить сценарий **inurest** для восстановления файлов из списка установки компонента **usr**.
4. Выполнить следующие действия для каждого набора файлов, указанного в файле **$FILESETLIST.new**:

   Примечание: В случае возникновения ошибки при выполнении любого из перечисленных шагов информация об этой ошибке записывается в файл **status**, а обработка набора файлов прекращается.

   a. Проверить, установлена ли в системе текущая или более старая версия набора файлов, а также установлены ли наборы файлов, перечисленные в файле *набор-файлов*.**namelist**. Если да, то экспортировать переменные среды **INSTALLED_LIST** и **MIGSAVE**. Такая операция называется *обновлением версии*.

   b. При установке обновления запустить сценарий *набор-файлов*.**post_u** (если он существует). Если сценарий *набор-файлов*.**post_u** не существует, то запустить *набор-файлов*.**post_i** (если он существует).

   c. Если существует файл *набор-файлов*.**cfgfiles**, то запустить сценарий **/usr/lib/instl/migrate_cfg** для обработки файлов конфигурации.

   d. Запустить **sysck** для добавления информации из файла *набор-файлов*.**inventory** в реестр программного обеспечения (SWVPD).

   e. Если существует файл ***набор-файлов*.tcb**, и в базе данных ODM **/usr/lib/objrepos/PdAt** установлен атрибут **tcb_enabled**, то запустить команду **tcbck** для добавления в систему информации о защищенной компьютерной базе.

   f. Если существует файл *набор-файлов*.**err**, то запустить сценарий **errupdate** для добавления шаблонов ошибок.

   g. Если существует файл *набор-файлов*.**trc**, то добавить шаблоны форматирования отчетов трассировки с помощью команды **trcupdate**.

   h. Если устанавливается обновление или существует файл ***рабочий-каталог*/*набор-файлов*.installed_list** то вызвать все сценарии вида *набор-файлов*.**odmdel** и *набор-файлов*.***.odmdel** для удаления ненужной информации из базы данных ODM.

   i. Вызвать команду **odmadd** для всех файлов *набор-файлов*.**odmadd** и *набор-файлов*.***.odmadd** для добавления информации в базу данных ODM.

   j. Если устанавливается обновление, то запустить сценарий *набор-файлов*.**config_u** (обновление конфигурации набора файлов), если он существует. В противном случае запустить сценарий *набор-файлов*.**config** (настройка набора файлов), если он существует.

   k. Указать в файле **status**, что текущий набор файлов успешно обработан.

5. Создать связи с управляющими файлами, необходимыми для удаления набора файлов, в каталоге **deinstl** пакета. Это могут быть следующие файлы:
   - **lpp.deinstal**
   - *набор-файлов*. **al**
   - *набор-файлов*. **inventory**

- *набор-файлов***. pre_d**
- *набор-файлов***. unpre_i**
- *набор-файлов***. unpre_u**
- *набор-файлов***. unpost_i**
- *набор-файлов***. unpost_u**
- *набор-файлов***. unodmadd**
- *набор-файлов***. unconfig**
- *набор-файлов***. unconfig_u**
- $SAVEDIR/*набор-файлов***. *.rodmadd**
- SAVEDIR/*набор-файлов***. *.unodmadd**

Алгоритмы работы стандартных сценариев аннулирования и очистки

В этом разделе описаны шаги, выполняемые командой **installp** при аннулировании обновлений и при очистке после неудачной установки наборов файлов. Стандартные сценарии **cleanup** и **reject** расположены в каталоге **/usr/lib/instl** и фактически представляют собой одну и ту же программу. Алгоритм работы этой программы незначительно отличается в случаях, когда она вызывается как **reject** и как **cleanup**. Если набор файлов состоит из компонентов **usr** и **root**, то компоненты **root** обрабатываются раньше компонентов **usr**.

1. Если выполняется операция аннулирования, то убедиться, что одновременно аннулируются все зависимые обновления.
2. Для каждого компонента пакета (например, **usr** или **root**) выполнить следующие действия:
   a. Присвоить значения переменным среды **INUTREE** (*U* для **usr** и *M* для **root**.) и **INUTEMPDIR**.
   b. Если в текущем каталоге (**INULIBDIR**) существует управляющий файл **reject**, то запустить сценарий **./lpp.reject**. В противном случае запустить сценарий по умолчанию **/usr/lib/instl/reject**.
3. Обновить реестр программного обеспечения (SWVPD).

Команда **installp** передает сценарию **reject** в первом параметре неопределенное значение, а во втором - полное имя файла со списком аннулируемых наборов файлов (ниже этот файл обозначается как **$FILESETLIST**).

В сценариях **cleanup** и **reject** задействован ряд дополнительных файлов.

Стандартные сценарии **cleanup** и **reject** выполняют следующие действия:

1. Выполнить следующие действия для каждого набора файлов из **$FILESETLIST**:
   a. Если вызван сценарий **cleanup**, определить по содержимому файла *имя-пакета***.s**, на каком шаге была прервана установка, и перейти к соответствующему шагу. Сценарий **cleanup** начнет работу только с того шага, на котором произошла ошибка при установке. Например, если ошибка произошла при выполнении сценария *Набор-файлов*.**post_i**, то очистка для этого набора файлов начнется с шага i, поскольку отмена более поздних операций установки не требуется.
   b. Аннулировать результаты настройки, выполненной во время установки:

      Если требуется аннулировать обновление, то вызвать сценарий *набор-файлов***.unconfig_u**, если он существует. Иначе вызвать сценарий *набор-файлов***.unconfig**, если он существует.
   c. Выполнить сценарии вида *набор-файлов***.*.unodmadd** и *набор-файлов***.unodmadd** для удаления записей ODM, добавленных во время установки.
   d. Выполнить сценарии вида *набор-файлов* **.*.rodmadd** и *набор-файлов***.rodmadd**, если они существуют, для восстановления записей ODM, удаленных во время установки.
   e. Если существует файл *набор-файлов***.undo.trc**, то выполнить сценарий **trcupdate**, который удаляет все изменения, внесенные в шаблоны отчетов трассировки при установке набора файлов.
   f. Если существует файл *набор-файлов***.undo.err**, то вызвать сценарий **errupdate**, который удаляет все изменения, внесенные в шаблоны ошибок во время установки.

g. Если существует файл *набор-файлов***.tcb**, и в базе данных ODM **/usr/lib/objrepos/PdAt** задан атрибут **tcb_enabled**, то удалить из системы информацию из защищенной компьютерной базы с помощью команды **tcbck**.

h. Если существует файл *набор-файлов***.inventory**, то выполнить сценарий **sysck**, который удаляет изменения, внесенные в базу данных программного обеспечения.

i. Аннулировать результаты операций настройки, выполненных после установки набора файлов:

Если требуется аннулировать обновление, то вызвать сценарий *набор-файлов***.unpost_u**, если он существует. Иначе вызвать сценарий *набор-файлов***.unpost_i**, если он существует.

j. Создать список установки (файл **master.al**) на основе файлов *набор-файла***.al**.

k. Добавить *набор-файлов* в файл **$FILESETLIST.new**.

2. Если существует файл **$INUTEMPDIR/master.al**, то выполнить следующие действия:
   a. Перейти в каталог **/** (**root**).
   b. Удалить все файлы, указанные в файле **master.al**.
3. Прочитать файл **$FILESETLIST.new** и выполнить следующие действия:
   a. С помощью программы **inurecv** восстановить все сохраненные файлы.
   b. Если требуется аннулировать обновление, то вызвать сценарий *набор-файлов***.unpre_u**, если он существует. Иначе вызвать сценарий *набор-файлов***.unpre_i**, если он существует.
   c. Удалить файлы управления установкой.
4. Удалить файл состояния *имя-пакета***.s**.

Алгоритм удаления программного обеспечения

В этом разделе описаны действия, выполняемые командой **installp** при удалении набора файлов. Если набор файлов состоит из компонентов **usr** и **root**, то компоненты **root** обрабатываются раньше компонентов **usr**.

1. Просмотреть список необходимого программного обеспечения и убедиться, что одновременно удаляются все зависимые наборы файлов.
2. Для каждого компонента пакета (например, **usr** или **root**) выполнить следующие действия:
   a. Присвоить нужные значения переменным среды INUTREE (U для **usr**, M для **root** и S для **share**) и INUTEMPDIR (рабочий каталог **installp** в файловой системе **/tmp**).
   b. Перейти в каталог **INULIBDIR**.
   c. Если в текущем каталоге существует файл **deinstal**, то запустить сценарий **./lpp.deinstal**. Если управляющий файл **deinstal** отсутствует в текущем каталоге, то запустить сценарий по умолчанию **/usr/lib/instl/deinstal**.
3. Удалить файлы, относящиеся к данному набору файлов.
4. Удалить всю информацию о наборе файлов, кроме информации хронологии, из реестра программного обеспечения (SWVPD).
5. Отменить регистрацию лицензионного соглашения на применение набора файлов.

Команда **installp** передает сценарию **deinstal** в качестве первого параметра полное имя файла со списком удаляемых наборов файлов. Этот файл в дальнейшем будет обозначаться как **$FILESETLIST**.

Сценарий **deinstal** выполняет следующие действия:

1. Выполнить следующие действия для каждого набора файлов, указанного в файле **$FILESETLIST**:
2. Если существует файл *набор-файлов***.unconfig_d**:

   Выполнить сценарий *набор-файлов***.unconfig_d**, который аннулирует все изменения конфигурации, изменения в базе данных ODM, изменения в шаблонах ошибок и информации трассировки, а также результаты работы сценариев установки обновлений и базового уровня набора файлов. Применять этот файл *не* рекомендуется.
3. Если файл *набор-файлов***.unconfig_d** не существует:

a. Для каждого обновления набора файлов выполнить следующие действия:
   - Выполнить все сценарии *набор-файлов***.unconfig_u** для аннулирования всех изменений, внесенных в конфигурацию при обновлении.
   - Выполнить все сценарии *набор-файлов***.*.unodmadd** и *набор-файлов***.unodmadd** для удаления записей ODM, добавленных во время обновления.
   - Выполнить все сценарии *набор-файлов***.*.rodmadd** и *набор-файлов***.rodmadd** для добавления записей ODM, удаленных во время обновления.
   - Если существует файл *набор-файлов***.undo.err**, то выполнить сценарий **errupdate**, который аннулирует все изменения в шаблонах ошибок.
   - Если существует файл *набор-файлов***.undo.trc**, то выполнить сценарий **trcupdate**, который аннулирует изменения в шаблонах отчетов трассировки.
   - Выполнить сценарий *набор-файлов***.unpost_u**, который аннулирует результаты настройки, выполненной после установки обновления.

b. Для базового уровня набора файлов выполнить следующие действия:
   - Выполнить сценарии вида *набор-файлов***.*.unodmadd** и *набор-файлов* **.unodmadd** для удаления записей ODM, добавленных во время установки.
   - Выполнить сценарии вида *набор-файлов***.*.rodmadd** и *набор-файлов***.rodmadd** для добавления в базу данных ODM записей, удаленных во время установки.
   - Если существует файл *набор-файлов***.undo.err**, то выполнить сценарий **errupdate**, который аннулирует все изменения в шаблонах ошибок.
   - Если существует файл *набор-файлов***.undo.trc**, то выполнить сценарий **trcupdate**, который аннулирует изменения в шаблонах отчетов трассировки.
   - Выполнить сценарий *набор-файлов***.unconfig_i**, чтобы аннулировать результаты настройки, выполненной во время установки.
   - Выполнить сценарий *набор-файлов***.unpost_i**, чтобы аннулировать результаты настройки, выполненной после установки.

4. Удалить из системы файлы данного набора и информацию о них.
5. Если файл *набор-файлов***.unconfig_d** не существует:
   a. Для каждого обновления набора файлов выполнить все сценарии *набор-файлов***.unpre_u**, чтобы аннулировать результаты настройки, выполненной до установки.
   b. Для базового уровня набора файлов выполнить все сценарии *набор-файлов***.unpre_i** для отмены настройки, выполненной до установки.
6. Удалить все пустые каталоги, оставшиеся после удаления набора файлов.

Примечание: Если во время выполнения сценария **deinstal** возникнет ошибка, информация о ней будет занесена в протокол, но выполнение сценария будет продолжено. В этом состоит отличие сценария deinstal от всех остальных сценариев команды installp, которые прекращают обработку набора файлов при возникновении ошибки. Поскольку отменить действие операции удаления нельзя, при возникновении ошибки наиболее разумное решение заключается в продолжении удаления.

Файл результатов установки

$INUTEMPDIR/status
В этом файле содержится список наборов файлов, которые должны были быть установлены или обновлены. Каждый набор файлов указан на отдельной строке.

Команда **installp** определяет по файлу **status** результаты обработки этих наборов файлов. Если вы будете поставлять собственные сценарии установки, они должны создавать файл **status** в правильном формате. Каждая строка файла **status** должна содержать следующую информацию:

код-результатанабор-файлов

Код результата	Значение
s	Набор файлов обработан успешно, можно обновить реестр программного обеспечения.
f	Сбой, требуется очистка
b	Сбой, очистка не требуется
i	Не выполнены условия установки, очистка не требуется
v	Набор файлов не прошел проверку **sysck**

Ниже приведен пример файла **status**, в котором указано, что наборы файлов **tcp.client** и **tcp.server** из пакета **bos.net** были установлены успешно, а набор файлов **nfs.client** установить не удалось.

```
s bos.net.tcp.client
s bos.net.tcp.server
f bos.net.nfs.client
```

Команды, выполняемые во время установки и обновления программного обеспечения

inucp При установке компонента root копирует файлы из каталога **/usr/lpp/*имя-пакета*/inst_root** в корневую файловую систему (/).

inulag Выполняет функцию интерфейса процедур управления лицензионными соглашениями.

inurecv
Восстанавливает сохраненные файлы в случае сбоя, а также при аннулировании обновлений (**installp -r**).

inurest Восстанавливает файлы с дистрибутивного носителя в системе в соответствии со списком установки.

inusave
Сохраняет файлы, указанные в списке установки, в каталоге сохранения программного продукта.

inuumsg
Выдает сообщения из каталога `inuumsg.cat` для устанавливаемого продукта.

ckprereq
Проверяет условия установки программного продукта, указанные в файле lpp_name, с учетом информации из реестра программного обеспечения (SWVPD).

sysck Проверяет информацию об архиве программного обеспечения во время установки и обновления.

Команда **sysck** находится в каталоге **/usr/bin**. Остальные перечисленные команды находятся в каталоге **/usr/sbin**.

Примеры применения этих команд можно найти в стандартном сценарии установки - **/usr/lib/instl/instal**.

Система контроля исходного кода

Система контроля исходного кода (SCCS) - это полная система команд, с помощью которой отдельные пользователи могут управлять файлами SCCS и отслеживать изменения в них. Допускается одновременное существование нескольких версий одного файла SCCS. Такая возможность полезна, например, при разработке проектов, когда требуется хранить множество версий больших файлов.

Команды SCCS поддерживают набор многобайтовых символов (MBCS).

Система SCCS - Введение

Команды SCCS предоставляют полный набор функций для создания, редактирования, преобразования и управления файлами SCCS. Файл SCCS - это любой текстовый файл, который применяется командами SCCS. В отличие от стандартных текстовых файлов, все файлы SCCS обозначаются префиксом **s.**.

> **Внимание:** Редактировать файлы SCCS следует только с помощью команд SCCS, в противном случае возможно повреждение файла.

Команды SCCS предназначены только для файлов SCCS. Если вы хотите ознакомиться со структурой файла SCCS, вы можете просмотреть его содержимое с помощью команды **pg** (или аналогичной). Однако напрямую изменять файл через текстовый редактор не следует.

Для того чтобы изменить текст файла SCCS, нужно сначала получить его версию для редактирования (например, с помощью команды SCCS **get**), а затем внести изменения в любом текстовом редакторе. После этого следует сохранить изменения с помощью команды **delta**. Файлы SCCS обладают уникальной структурой, которая позволяет хранить различные версии файла и управлять доступом к ним.

Файл SCCS состоит из трех частей:

- Таблица поправок
- Флаги доступа и контроля
- Тело файла

Таблица поправок в файле SCCS

Файловая система SCCS не создает каждую новую версию в виде отдельного файла, а сохраняет лишь внесенные в каждую версию изменения, которые называют *поправками*. Все поправки к данному файлу SCCS заносятся в его таблицу поправок.

Каждая запись в таблице изменений содержит сведения о том, когда, кем и для чего была создана данная поправка. Каждая поправка снабжена не более чем 4-значным SID - идентификационным номером SCCS. Первая цифра - это выпуск, вторая - уровень, третья - ветвь, а четвертая - порядковый номер.

Пример номера SID:

```
SID = 1.2.1.4
```

т.е. выпуск 1, уровень 2, ветвь 1, порядковый номер 4.

Ни одна из цифр SID не должна быть равна 0, поэтому SID вида 2.0 или 2.1.2.0 недопустимы.

Каждой новой поправке по умолчанию присваивается следующий по порядку SID. При создании новой версии учитываются все предыдущие поправки. Как правило, поправки вносятся в файл SCCS последовательно, поэтому они идентифицируются только выпуском и уровнем. Однако в принципе возможно и параллельное внесение поправок (ветвление). В этом случае совокупность поправок подразделяется на "ствол" (SID состоит из выпуска и уровня) и одну или несколько "ветвей" (SID состоит из выпуска, уровня, ветви и порядкового номера). Внутри ветви номера выпуска и уровня у всех поправок одинаковы, а порядковые номера различны.

> **Примечание:** Версии файла, построенные на поправках данной ветви, не учитывают стволовые поправки, появившиеся после точки ветвления.

Флаги доступа и контроля в файле SCCS

После таблицы поправок в файле SCCS расположен список флагов, начинающихся со знака @. Флаги определяют различные опции доступа и контроля в файлах SCCS. Ниже приведены некоторые их функции:

- Назначение пользователей, которые имеют право редактировать файлы
- Запрет редактирования определенных версий файла
- Разрешение совместного редактирования файла
- Перекрестное согласование изменений файла

Тело файла SCCS

В теле файла SCCS содержится текст всех его версий. Следовательно, тело не является обычным текстовым файлом. Каждый фрагмент текста ограничен управляющими символами, которые указывают, какой поправкой он был создан или удален. Когда система SCCS создает конкретную версию файла, управляющие символы позволяют определить соответствие между фрагментами текста и поправками. Затем на основе выбранных фрагментов создается данная версия.

Стандарты флагов и параметров SCCS

В этом разделе рассмотрены флаги команд SCCS.

Для большинства команд SCCS предусмотрены два типа параметров:

Параметр	Описание
флаги	Флаг состоит из знака минус (-), строчной латинской буквы и необязательного числа. Флаги служат для управления командами.
Файл или *каталог*	Эти параметры указывают, к какому файлу (файлам) следует применять команду. Если в качестве аргумента задано имя каталога, то команда относится ко всем файлам SCCS в этом каталоге.

Имя каталога или файла не может начинаться со знака минус. Если вы укажете только этот знак, то команда будет считывать стандартный ввод или ввод с клавиатуры до тех пор, пока не достигнет конца файла. Это оказывается удобным при работе с конвейерами, посредством которых процессы взаимодействуют друг с другом.

Все заданные в команде флаги относятся ко всем файлам, указанным в командной строке, и обрабатываются до всех остальных параметров команды. Положение флага в командной строке не имеет значения, остальные же параметры обрабатываются по порядку слева направо. В некоторые файлы SCCS включены флаги, которые определяют порядок выполнения определенных команд для данного файла. Более подробные сведения о флагах в заголовках файлов SCCS приведены в описании команды **admin**).

Создание, редактирование и обновление файлов SCCS

Для создания, редактирования и обновления файлов SCCS предназначены команды **admin**, **get** и **delta**.

Создание файла SCCS

admin

Создает новый файл SCCS или изменяет уже существующий.

- Для создания пустого файла SCCS с именем `s.test.c` введите следующую команду:

  ```
  admin -n s.test.c
  ```

  (Команда **admin** с флагом **-n** создает пустой файл SCCS).

- Для преобразования текстового файла в файл SCCS введите следующую команду:

  ```
  admin -itest.c s.test.c
  В файле отсутствуют ключевые идентификационные слова SCCS (cm7)

  ls
  s.test.c test.c
  ```

  Если запустить команду **admin** с флагом **-i**, то на основе указанного файла будет создана поправка 1.1. После этого, во избежание ошибок в командах SCCS, исходный текстовый файл следует переименовать (он будет служить резервной копией):

  ```
  mv test.c back.c
  ```

  Сообщение `В файле отсутствую ключевые идентификационные слова SCCS (cm7)` - не является сообщением об ошибке.

- Для запуска файла `test.c` выпуска 3.1 введите команду **admin** с флагом **-r**:

```
admin -itest.c -r3 s.test.c
```

Изменение файла SCCS

Внимание: Редактировать файлы SCCS следует только с помощью команд SCCS, в противном случае возможно повреждение файлов.

get

Вызывает заданную версию файла SCCS для редактирования или компиляции.

1. Для изменения файла SCCS введите команду **get** с флагом **-e**:

```
get -e s.test.c
1.3
новая поправка 1.4
67 строк

ls
p.test.c s.test.c test.c
```

Команда **get** создает два новых файла: `p.test.c` и `test.c`.. `test.c` - это файл для редактирования, а `p.test.c` - временный, для отслеживания версий файлов SCCS. Редактировать его нельзя и после обновления файла SCCS он будет удален. Обратите внимание также и на то, что команда **get** выводит SID версии файла для редактирования, SID новой поправки и число строк в файле.

2. Файл `test.c` можно изменить в любом текстовом редакторе, например:

```
ed test.c
```

Теперь вы можете начинать работу с файлом. Изменять его можно с любой частотой. Изменения никак не отразятся на самом файле SCCS до тех пор, пока он не будет обновлен.

3. Для того чтобы отредактировать определенную версию файла SCCS, введите команду **get** с флагом **-r**:

```
get -r1.3 s.test.c
1.3
67 строк

get -r1.3.1.4 s.test.c
1.3.1.4
50 строк
```

Обновление файла SCCS

delta

Вносит в текст файла SCCS изменения поправки.

1. Для того чтобы обновить файл SCCS и создать новую поправку, введите команду **delta**:

```
$delta s.test.c
Введите комментарии или пустую строку, а затем EOF.
```

2. Команда **delta** автоматически запрашивает у вас комментарии для каждой новой поправки. Например, введите комментарий, а затем дважды нажмите клавишу Enter:

```
Отсутствуют id ключевые слова (cm7)
1.2
5 строк добавлено
6 строк удалено
12 строк - без изменений
```

Команда **delta** записывает в файл s.prog.c изменения, которые вы внесли в файл `test.c`. Команда **delta** сообщает, что SID новой версии равен 1.2, и что по сравнению с предыдущей версией, в файл было добавлено 5 строк, удалено 6 строк, а 12 строк остались без изменений.

Управление доступом и отслеживание изменений в файлах SCCS

Команда и файловая система SCCS в основном применяются для управления доступом к файлам и отслеживания изменений в них (что именно, почему и кем было изменено).

Управление доступом к файлам SCCS

Файловые системы SCCS позволяют управлять тремя типами доступа:

- Доступ к файлам
- Доступ пользователей
- Доступ к версии

Управление доступом к файлам

Каталогам, в которых будут храниться файлы SCCS следует при создании присваивать код прав доступа 755 (права на чтение, запись и выполнение для владельца, а также права на чтение и выполнение для членов группы и прочих пользователей). Сами файлы SCCS следует создавать с кодом 444 (права только на чтение). При изменять файл SCCS с помощью обычных (не SCCS) команд сможет только владелец файлов. Если у членов группы есть права доступа на запись в файлы SCCS, то и для каталогов также следует указывать права доступа на запись для группы.

Управление доступом пользователей

Команда **admin** с флагом **-a** позволяет определить группу пользователей, которым разрешено изменять файл SCCS. С помощью этого флага можно также задать имя или номер группы.

Управление доступом к версиям

Команда **admin** позволяет запретить доступ к различным версиям файла путем вызова команды **get** с помощью флагов заголовка.

-fc
Указывает последний из доступных выпусков

-ff
Указывает самый ранний из доступных выпусков

-fl
Блокирует доступ к определенному выпуску

Отслеживание изменений в файле SCCS

Существует три способа отслеживания изменений, вносимых в файл SCCS:

- Комментарии к каждому обновлению
- Номера Запросов на изменение (MR)
- Команды SCCS.

Отслеживание изменений с помощью комментариев к обновлениям

После обновление файла SCCS появляется приглашение для ввода соответствующего комментария. Длина комментариев не должна превышать 512 символов. Их можно редактировать с помощью команды **cdc**.

cdc
Изменяет комментарий к обновлению.

Команда **get** с флагом **-l** позволяет просмотреть таблицу обновлений и комментариев к ним для любых версий файла. Кроме комментариев, в этой таблице автоматически регистрируется дата и время последнего

изменения файла, фактический ИД пользователя на момент изменения, номера предыдущего и текущего обновлений, а также все связанные с обновлением номера MR.

Отслеживание изменений с помощью номеров запросов на изменение

Команда **admin** с флагом **-fv** при создании каждого обновления запрашивает номера MR. Флаг **-fv** позволяет задать программу проверки номеров MR при создании нового обновления файла SCCS. Если программа проверки вернет значение, отличное от нуля, то обновление будет запрещено.

Программа проверки номеров MR создается пользователем. Можно создать программу, которая будет отслеживать изменения файла SCCS и индексировать их в какой-либо базе данных или системе контроля.

Отслеживание изменений с помощью команд SCCS

`sccsdiff`
Сравнивает два файла SCCS и выводит информацию об их отличиях в стандартный вывод

Команда **delta** с флагом **-p** действует аналогично команде **sccsdiff** при обновлении файла. Эти две команды позволяют отслеживать изменения, внесенные в файл в промежутке между версиями.

`prs`
Форматирует и записывает определенные фрагменты файла SCCS в стандартный вывод

Эта команда показывает различия между двумя версиями файла.

Обнаружение и исправление повреждений в файлах SCCS

Для обнаружения и исправления повреждений в файлах SCCS служит команда **admin**.

Процедура

1. Регулярно проверяйте наличие повреждений в файлах SCCS. Если файл SCCS изменяется не с помощью специальных команд SCCS, а каким-либо иным способом, то существует возможность повреждения этого файла. Файловая система SCCS обнаруживает повреждения путем проверки контрольной суммы и сравнения ее со значением из таблицы. Для проверки файлов введите команду **admin** с флагом **-h**, перечислив в ней все файлы и каталоги SCCS:

   ```
   admin -h s.файл-1 s.файл-2 ...
   ```

   ИЛИ

   ```
   admin -h каталог-1 каталог-2 ...
   ```

   Если команда **admin** обнаруживает файл SCCS, у которого контрольная сумма не совпадает с указанной в заголовке, то появляется следующее сообщение:

   ```
   ERROR [s.filename]:
   1255-057 Файл поврежден. (сов)
   ```

2. Если файл был поврежден, попробуйте исправить его или восстановить из резервной копии. Обнаружить повреждение файла с помощью команды **admin** можно только в том случае, если контрольная сумма еще не обновлена.

   > **Примечание:** После запуска для поврежденного файла команды **admin** с флагом **-z**, обнаружение других повреждений этого файла станет невозможным.

3. После исправления файла запустите команду **admin**, указав в ней флаг **-z** и имя файла:

   ```
   admin -z s.file1
   ```

Список дополнительных команд SCCS

Для работы с файлами SCCS предназначены следующие команды SCCS:

Внимание: при выполнении для файлов SCCS команд, отличных от команд SCCS, эти файлы могут быть повреждены.

Команда	Описание
rmdel	Удаляет из файла SCCS самую последнюю дельту ветви.
sact	Показывает текущее состояние редактирования файла SCCS.
sccs	Административная программа для системы SCCS. Команда **sccs** состоит из набора псевдокоманд, выполняющих большинство служебных функций SCCS.
sccshelp	Выдает справку по командам и сообщения об ошибках SCCS.
unget	Отменяет действие предыдущей команды **get -e**.
val	Проверяет, совпадает ли подсчитанная контрольная сумма файла SCCS с указанной в заголовке.
vc	Подставляет присвоенные значения вместо ключевых слов.
what	Выполняет поиск по шаблону в системном файле и выводит текст, который следует за шаблоном.

Функции, примеры программ и библиотеки

В данном разделе объясняется, что такое функции, как их использовать, и где они хранятся.

Для экономии памяти и более эффективной компоновки программ функции хранятся в виде библиотек. *Библиотека* - это файл данных, содержащий копии отдельных файлов и управляющую информацию, которая позволяет получать доступ к отдельным файлам. Библиотеки располагаются в каталогах **/usr/ccs/lib** и **/usr/lib**. По общепринятому соглашению имена большинства из них задаются в виде **lib***имя***.a**, где *имя* задает конкретную библиотеку.

Все директивы включения файлов должны задаваться в начале первого компилируемого файла, обычно в разделе объявлений перед **main( )**, и до использования библиотечных функций. Например, для включения файла **stdio.h** используйте следующую директиву:

```
#include <stdio.h>
```

Для вызова стандартных функций языка C из библиотеки **libc.a** не нужно выполнять никаких специальных действий. Команда **cc** автоматически находит эту библиотеку для функций, вызываемых в данной программе. Однако если вы вызываете функцию из другой библиотеки, необходимо сообщить компилятору имя этой библиотеки. Если в вашей программе используются функции из библиотеки **lib***имя***.a**, то компилируйте программу с флагом **-l***имя* (L в нижнем регистре). Для компиляции программы `myprog.c`, в которой используются функции из библиотеки **libdbm.a** необходимо ввести команду

```
cc myprog.c -ldbm
```

Можно задать несколько флагов **-l** (L в нижнем регистре). Флаги обрабатываются в том порядке, в котором указаны.

Если вы планируете использовать функцию из библиотеки Berkeley Compatibility Library, то необходимо *сначала* подключить библиотеку **libbsd.a**, и только потом - библиотеку **libc.a**, как показано в следующем примере:

```
cc myprog.c -lbsd
```

При возникновении ошибки большинство функций возвращают значение -1 и записывают код ошибки во внешнюю переменную **errno**. Переменная **errno** объявляется в файле **sys/errno.h**, кроме того, в этом файле определяются константы для всех возможных исключительных ситуаций.

В данном руководстве все системные вызовы описываются как *функции*; предполагается, что они хранятся в библиотеке **libc.a**. Программный интерфейс системных вызовов аналогичен интерфейсу функций. В программах на языке C системный вызов - это просто вызов функции. Реальное различие между системным вызовом и функцией заключается в типе выполняемой ими операции. Когда программа выполняет системный вызов, переключается режим защиты домена, и вызываемая процедура получает доступ к специальной информации ядра операционной системы. При этом процедура работает в режиме ядра и

выполняет задачу вместо программы. Таким образом, доступ к специальной системной информации разрешен ограниченному набору предопределенных процедур, действиями которых можно управлять.

Примечание:

1. Ниже приведен список устаревших функций wString 64-разрядной библиотеки **libc.a**. Здесь же приводятся соответствующие функции 64-разрядной библиотеки **libc.a**. Список функций 32-разрядной библиотеки **libc.a** приведен в описании функции wstring.

   ```
   32-разрядная  64-разрядная

   wstrcat wcscat
   wstrchr wcschr
   wstrcmp wcscoll
   wstrcpy wcscpy
   wstrcspn wcscspn
   wstrdup                    Недоступна и не имеет аналогов
                              в 64-разрядной libc.a
   wstrlen wcslen
   wstrncat wcsncat
   wstrncpy wcsncpy
   wstrpbrk wcspbrk
   wstrrchr wcsrchr
   wstrspn wcsspn
   wstrtok wcstok
   ```

2. В начале любой программы, работающей с многобайтовыми, широкими или определяемыми локалью символами, должна вызываться функция **setlocale**.
3. Для того чтобы обеспечить целостность данных, при создании программ с несколькими нитями следует применять реентерабельные функции.

Понятия, связанные с данным:

“Инструменты и утилиты” на стр. 1
В этом разделе приведен обзор инструментов и утилит, предназначенных для разработки программ на языке C.

Информация, связанная с данной:

Список служб управления данными о времени

Обзор файлов заголовков

itrunc

printf

scanf

setlocate

sqrt

128-разрядные числа двойной точности

Операционная система AIX поддерживает 128-разрядные числа двойной точности, обеспечивающие большую точность, чем применяемые по умолчанию 64-разрядные числа двойной точности. В 128-разрядных данных количество значащих цифр равно 31 (в 64-разрядных данных - только 17). Однако дополнительные значащие цифры повышают только точность (длину дробной части) чисел, но не их диапазон по абсолютной величине.

Работе со 128-разрядными данными двойной точности посвящены следующие разделы:

- Компиляция программ, использующих 128-разрядные данные двойной точности
- Соответствие стандарту IEEE 754
- Реализация 128-разрядного формата двойной точности
- Значения для числовых макроподстановок

Компиляция программ, использующих 128-разрядные данные двойной точности

Для компиляции программ на языке C, в которых применяются 128-разрядные числа двойной точности, служит команда **xlc128**. Это разновидность команды **xlc**, поддерживающая 128-разрядные данные. Команда **xlc** поддерживает только 64-разрядные данные.

В стандартной библиотеке языка C **libc128.a** существуют аналоги процедур библиотеки **libc.a**, поддерживающие числа двойной точности. При компиляции приложений, использующих 64-разрядные данные двойной точности, компоновку следует выполнять с библиотекой **libc.a**. Приложения же, использующие 128-разрядные значений двойной точности, следует компоновать с обеими библиотеками **libc128.a** и **libc.a**, причем в списке просмотра библиотека **libc128.a** должна стоять перед библиотекой **libc.a**.

Соответствие стандарту IEEE 754

Функции для работы с 64-разрядными числами двойной точности полностью совместимы со стандартом IEEE 754, в отличие от функций для работы с 128-разрядными числами. Если вы хотите, чтобы приложения соответствовали стандарту IEEE 754, применяйте в них 64-разрядный формат данных.

128-разрядная реализация не соответствует стандарту IEEE в следующем:

- Поддерживает только режим "округления до ближайшего". Если изменить в приложении режим округления, то результаты становятся непредсказуемыми.
- Не полностью поддерживает специальные числа IEEE NaN и INF.
- Не поддерживает флаги состояния IEEE для переполнения, потери значимости и других ошибок. Эти флаги не применяются в реализации 128-разрядных чисел двойной точности.
- Для 128-разрядных чисел двойной точности не поддерживаются следующие математические API: **atanhl**, **cbrtl**, **copysignl**, **exp2l**, **expm1l**, **fdiml**, **fmal**, **fmaxl**, **fminl**, **hypotl**, **ilogbl**, **llrintl**, **llroundl**, **log1pl**, **log2l**, **logbl**, **lrintl**, **lroundl**, **nanl**, **nearbyintl**, **nextafterl**, **nexttoward**, **nexttowardf**, **nexttowardl**, **remainderl**, **remquol**, **rintl**, **roundl**, **scalblnl**, **scalbnl**, **tgammal** и **truncl**.

Реализация 128-разрядного формата двойной точности

128-разрядное число двойной точности представляет собой упорядоченную пару 64-разрядных чисел двойной точности. Первый элемент этой пары содержит старшую, а второй - младшую часть числа. Длинное значение двойной точности является суммой этих 64-разрядных чисел.

Каждое из этих двух 64-разрядных чисел является числом двойной точности с плавающей точкой со знаком, порядком и мантиссой. Обычно модуль младшего числа меньше последнего разряда старшего, поэтому их значения не перекрываются и мантисса младшего числа повышает точность старшего числа.

Такое представление чисел обладает некоторыми особенностями, которые следует принять во внимание:

- Диапазон модулей (порядков) 128-разрядных чисел совпадает с диапазоном модулей 64-разрядных чисел. При переходе к 128-разрядному формату повышается только точность, но не диапазон абсолютной величины.
- По мере уменьшения модуля числа дополнительная точность, задаваемая младшей частью, также снижается. Если число попадает в диапазон слишком малых чисел, то данное представление обеспечивает не большую точность, чем 64-разрядный формат данных с двойной точностью.
- Фактическое количество бит точности может меняться. Если младшая часть числа намного меньше последнего разряда старшей части, то подразумевается, что между мантиссами старшего и младшего чисел находятся значащие биты (все нули или все единицы). Алгоритмы, базирующиеся на фиксированной длине мантиссы, могут оказаться ошибочными при работе с 128-разрядными числами двойной точности.

Значения для числовых макроподстановок

Из-за способа хранения длинных чисел двойной точности макросам могут соответствовать различные числа. При использовании 128-разрядного представления длинных чисел двойной точности значения следующих макросов из файла **values.h**, обязательных в стандарте языка C, не определены:

- Число разрядов в мантиссе (**LDBL_MANT_DIG**)
- Эпсилон (**LBDL_EPSILON**)
- Максимальное конечное представимое число (**LDBL_MAX**)

Число разрядов в мантиссе

Длина мантиссы не фиксирована, но для правильно отформатированных чисел (за исключением слишком малых) минимально возможное число разрядов равно 106. Поэтому значение макроса **LDBL_MANT_DIG** - 106.

Эпсилон

В стандарте ANSI C значение эпсилон определено как разность между наименьшим представимым числом, большим единицы, и единицей, т. е. *b***(1-*p*), где *b* - это основание системы исчисления (2), а *p* - количество разрядов в числе в таком представлении. По этому определению необходимо, чтобы длина основания *b* была фиксированной, а это неверно для 128-разрядных чисел двойной точности.

Ниже приведено наименьшее представимое число, большее единицы:

```
0x3FF0000000000000, 0x0000000000000001
```

Разность между этим значением и единицей равна:

```
0x0000000000000001, 0x0000000000000000
0.49406564584124654417656879286822 13E-323
```

Так как в 128-разрядных числах точность обычно составляет не менее 106 разрядов, то минимальное значение *p* равно 106. Таким образом, по формуле *b***(1-*p*), 2**(-105) получается следующее значение:

```
0x3960000000000000, 0x0000000000000000
0.24651903288156618919116517665087070E-31
```

Оба значения подходят под определение значения эпсилон в соответствии со стандартом C. В функциях, использующих длинные числа двойной точности, будет применено второе значение, потому что оно лучше характеризует точность, достигаемую 128-разрядной реализацией.

Максимальное длинное число двойной точности

Значение макроса **LDBL_MAX** задает максимальное 128-разрядное число двойной точности, остающееся неизменным при умножении на 1.0. Это также максимальное конечное число, которое можно получить в результате простейших операций, такими как умножение и деление:

```
0x7FEFFFFFFFFFFFFF, 0x7C9FFFFFFFFFFFFF
0.17976931348623159077293051907890025 75e+309
```

Список функций для работы с символами

Функции и макрокоманды символьной обработки выполняют тестирование и преобразование символов ASCII.

Операции, выполняемые этими функциями и макрокомандами, подразделяются на три типа:

- Определение типа символа
- Преобразование символов
- Посимвольный ввод-вывод

Раздел Определение типа символа иллюстрирует некоторые процедуры работы с символами.

Определение типа символа

Следующие функции и макрокоманды определяют тип символа. Результаты применения функций, работающих со знаками препинания и алфавитно-цифровыми символами, а также функций, различающих регистр, зависят от текущей последовательности упорядочения.

Библиотека **ctype** содержит следующие функции:

isalpha
Является ли данный символ алфавитным?

isalnum
Является ли данный символ алфавитно-цифровым?

isupper
Является ли данный символ прописной буквой?

islower
Является ли данный символ строчной буквой?

isdigit
Является ли данный символ цифрой?

isxdigit
Является ли данный символ шестнадцатеричной цифрой?

isspace
Является ли данный символ пробелом?

ispunct
Является ли данный символ знаком препинания?

isprint
Является ли данный символ печатаемым, включая пробел?

isgraph
Является ли данный символ печатаемым, исключая пробел?

iscntrl
Является ли данный символ управляющим?

isascii
Является ли данный символ ASCII-кодом?

Преобразование символов

Библиотека **conv** содержит следующие функции:

toupper
Преобразует строчную букву в прописную

_toupper
(Макрокоманда) Преобразует строчную букву в прописную

tolower
Преобразует прописную букву в строчную

_tolower
(Макрокоманда) Преобразует прописную букву в строчную

toascii
Преобразует целое число в символ ASCII

Посимвольный ввод-вывод

getc, fgetc, getchar, getw
Извлекает символ или слово из входного потока

putc, putchar, fputc, putw
Помещают символ или слово в поток

Список функций создания исполняемых программ

Средства создания выполняемой программы включают группу функций и команд.

Эти команды и функции позволяют создавать, компилировать и отлаживать файлы с исходным кодом программы.

Функция	Описание
_end, **_text**, **_edata**	Определяют последнее расположение программы
confstr	Определяет текущее значение указанной системной переменной, заданной в виде строки.
getopt	Получает значения флагов из массива аргументов.
ldopen, **ldaopen**	Открывает общий объектный файл
ldclose, **ldaclose**	Закрывает общий объектный файл
ldahread	Считывает заголовок архива из элемента архивного файла.
ldfhread	Считывает заголовок общего объектного файла.
ldlread, **ldlinit**, **ldlitem**	Считывает и обрабатывает записи номеров строк функции общего объектного файла
ldshread, **ldnshread**	Считывает заголовок раздела общего объектного файла
ldtbread	Считывает запись из таблицы имен общего объектного файла
ldgetname	Получает символьное имя из записи таблицы или из таблицы строк
ldlseek, **ldnseek**	Переходит к записям номеров строк раздела общего объектного файла
ldohseek	Переходит к необязательному заголовку общего объектного файла
ldrseek, **ldnrseek**	Переходит к информации о перемещении раздела общего объектного файла
ldsseek, **ldnsseek**	Переходит к разделу общего объектного файла
ldtbseek	Переходит к таблице имен общего объектного файла
ldtbindex	Возвращает индекс записи таблицы имен общего объектного файла
load	Загружает объектный модуль и связывает его с текущим процессом
unload	Выгружает объектный файл
loadbind	Выполняет динамическое преобразование имен в модуле
loadquery	Возвращает информацию об ошибках функции **load** или **exec**. Кроме этого, выдает список загруженных объектных файлов, связанных с текущим процессом
monitor	Запускает и останавливает профайлер программы
nlist	Получает записи из списка имен
regcmp, **regex**	Компилирует программу и подставляет шаблоны регулярных выражений
setjmp, **longjmp**	Сохраняет адрес программы
sgetl, **sputl**	Получает численные данные большого объема в машинно-независимой форме
sysconf	Выводит текущее значение заданного системного ограничения или опции

Список функций для работы с файлами и каталогами

В системе предусмотрены функции создания файлов, чтения и записи данных в файлы, а также функции задания ограничений и структуры файловой системы.

Большинство этих функций реализованы в виде команд. У вас есть возможность создать на основе таких функций собственные команды и утилиты, которые можно использовать как в процессе разработки программ, так и в самих программах.

В системе предусмотрены следующие функции:

Функции для работы с файлами

access, accessx или faccessx
Возвращают информацию о правах доступа к файлу

fclear
Удаляет данные из файла

fcntl, dup или dup2
Команды для работы с дескрипторами открытых файлов

fsync
Записывает внесенные в файл изменения на диск

getenv
Возвращает значение переменной среды

getutent, getutid, getutline, putuline, setutent, endutent или utmpname
Предназначены для доступа к записям файла utmp

getutid_r, getutline_r, pututline_r, setutent_r, endutent_r или utmpname_r
Предназначены для доступа к записям файла utmp

lseek и llseek
Изменяют смещение указателя в открытом файле

lockfx, lockf или flock
Предназначены для работы с блокировками дескрипторов открытых файлов

mknod или mkfifo
Позволяют создать обычный файл, специальный файл и файл FIFO

mktemp или mkstemp
Создают файл с уникальным именем

open, openx или creat
Создают файл и возвращает его дескриптор

pclose
Закрывает открытый канал

pipe
Создает канал для связи между процессами

popen
Инициализирует канал для связи с процессом

pathconf, fpathconf
Возвращают информацию об операциях, которые можно выполнить над файлом

putenv
Устанавливает значение переменной среды

read, readx, readv, readvx
Считывают данные из файла или с устройства

rename
Переименовывает каталог или файл в файловой системе

statx, stat, fstatx, fstat, fullstat, fullstat
Возвращают информацию о состоянии файла

tmpfile
Создает временный файл

tmpnam или tempnam
Создают имя временного файла

truncate, ftruncate
Усекают файл

umask
Получает и устанавливает маску создания файла

utimes или utime
Устанавливают права доступа к файлу и время изменения файла

write, writex, writev, writevx
Записывают данные в файл или на устройство

Функции работы с каталогами

chdir
Позволяет перейти в другой каталог

chroot
Изменяет текущий корневой каталог

getwd, getcwd
Возвращает путь к текущему каталогу

glob
Позволяет получить список путей к доступным файлам

globfree
Освобождает память, связанную с параметром *pglob*

link
Создает новую запись каталога для существующего файла

mkdir
Создает каталог

opendir, readdir, telldir, seekdir, rewinddir, closedir
Выполняют различные операции над каталогами

readdir_r
Считывает данные из каталога

rmdir
Удаляет каталог

scandir, alphasort
Позволяют выполнять поиск в каталоге

readlink
Считывает содержимое символьной связи

remove
Удаляет файл с указанным именем

symlink
Создает символьную связь с файлом

unlink
Удаляет запись каталога

Управление файловыми системами

confstr
Возвращает текущее значение системной переменной с заданным именем

fscntl
Предназначена для управления файловой системой

getfsent, getfsspec, getfsfile, getfstype, setfsent или endfsent
Позволяют получать информацию о файловой системе

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, endvfsent
Позволяют получить информацию о записях виртуальной файловой системы

mnctl
Возвращает информацию о состоянии монтирования

quotactl
Позволяет изменять ограничения на объем дисковой памяти

statfs, fstatfs
Позволяют получить информацию о состоянии файловой системы, в которой расположен файл

sysconf
Позволяет получить текущие значения установленных в системе опций и ограничений

sync
Обновляет на диске данные всех файловых систем

umask
Получает и устанавливает маску создания файла

vmount
Монтирует файловую систему

umount, uvmount
Удаляют виртуальную файловую систему из дерева каталогов

Список вектор-векторных функций для FORTRAN BLAS уровня 1

Уровень 1: для работы с векторами предназначены следующие функции:

Функция	Описание
SDOT, **DDOT**	Вычисляет скалярное произведение двух векторов
CDOTC, **ZDOTC**	Вычисляет комплексное скалярное произведение вектора, сопряженного с первым из заданных векторов, и второго вектора
CDOTU, **ZDOTU**	Вычисляет комплексное скалярное произведение двух векторов
SAXPY, **DAXPY**, **CAXPY**, **ZAXPY**	Умножают вектор на число и складывают его с другим вектором
SROTG, **DROTG**, **CROTG**, **ZROTG**	Рассчитывает поворот плоскости Гивенса
SROT, **DROT**, **CSROT**, **ZDROT**	Поворот плоскости
SCOPY, **DCOPY**, **CCOPY**, **ZCOPY**	Копирует вектор *X* в *Y*
SSWAP, **DSWAP**, **CSWAP**, **ZSWAP**	Меняет местами данные векторов *X* и *Y*
SNRM2, **DNRM2**, **SCNRM2**, **DZNRM2**	Вычисляет евклидову норму *N*-мерного вектора из *X*() с приращением *INCX*
SASUM, **DASUM**, **SCASUM**, **DZASUM**	Вычисляет сумму абсолютных значений компонентов вектора
SSCAL, **DSCAL**, **CSSCAL**, **CSCAL**, **ZDSCAL**, **ZSCAL**	Умножает вектор на константу
ISAMAX, **IDAMAX**, **ICAMAX**, **IZAMAX**	Выдает индекс максимального по модулю элемента
SDSDOT	Складывает скалярное произведение двух векторов и константу
SROTM, **DROTM**	Выполняет модифицированное преобразование Гивенса
SROTMG, **DROTMG**	Рассчитывает модифицированное преобразование Гивенса

Список матрично-векторных функций для FORTRAN BLAS уровня 2

Уровень 2: предусмотрены следующие матрично-векторные функции:

Функция	Описание
SGEMV, **DGEMV**,**CGEMV**, **ZGEMV**	Матрично-векторные операции над матрицами общего вида
SGBMV, **DGBMV**, **CGBMV**, **ZGBMV**	Матрично-векторные операции над ленточными матрицами
CHEMV, **ZHEMV**	Матрично-векторные операции над эрмитовыми матрицами
CHBMV, **ZHBMV**	Матрично-векторные операции над ленточными эрмитовыми матрицами
CHPMV,**ZHPMV**	Матрично-векторные операции над упакованными эрмитовыми матрицами
SSYMV , **DSYMV**	Матрично-векторные операции над симметричными матрицами
SSBMV , **DSBMV**	Матрично-векторные операции над симметричными ленточными матрицами
SSPMV , **DSPMV**	Матрично-векторные операции над упакованными симметричными матрицами
STRMV, **DTRMV**, **CTRMV**, **ZTRMV**	Матрично-векторные операции над треугольными матрицами
STBMV, **DTBMV**, **CTBMV**, **ZTBMV**	Матрично-векторные операции над треугольными ленточными матрицами
STPMV, **DTPMV**, **CTPMV**, **ZTPMV**	Матрично-векторные операции над упакованными треугольными матрицами
STRSV, **DTRSV**, **CTRSV**, **ZTRSV**	Решение систем уравнений
STBSV, **DTBSV**, **CTBSV**, **ZTBSV**	Решение систем уравнений
STPSV, **DTPSV**, **CTPSV**, **ZTPSV**	Решение систем уравнений
SGER, **DGER**	Операции ранга 1
CGERU, **ZGERU**	Операции ранга 1
CGERC,**ZGERC**	Операции ранга 1
CHER, **ZHER**	Операции над эрмитовыми матрицами ранга 1
CHPR,**ZHPR**	Операции над эрмитовыми матрицами ранга 1
CHPR2,**ZHPR2**	Операции над эрмитовыми матрицами ранга 2
SSYR, **DSYR**	Операции над симметричными матрицами ранга 1
SSPR, **DSPR**	Операции над симметричными матрицами ранга 1
SSYR2 , **DSYR2**	Операции над симметричными матрицами ранга 2
SSPR2 ,**DSPR2**	Операции над симметричными матрицами ранга 2

Список функций для работы с матрицами для FORTRAN BLAS уровня 3

Уровень 3: для работы с матрицами предназначены следующие функции:

Функция	Описание
SGEMM, **DGEMM**, **CGEMM**, **ZGEMM**	Операции над матрицами общего вида
SSYMM, **DSYMM**,**CSYMM**, **ZSYMM**	Операции над симметричными матрицами
CHEMM,**ZHEMM**	Операции над эрмитовыми матрицами
SSYRK, **DSYRK**,**CSYRK**, **ZSYRK**	Операции над симметричными матрицами ранга k
CHERK, **ZHERK**	Операции над эрмитовыми матрицами ранга k
SSYR2K, **DSYR2K**, **CSYR2K**, **ZSYR2K**	Операции над симметричными матрицами ранга 2k
CHER2K,**ZHER2K**	Операции над эрмитовыми матрицами ранга 2k
STRMM, **DTRMM**, **CTRMM**, **ZTRMM**	Операции над треугольными матрицами
STRSM, **DTRSM**, **CTRSM**, **ZTRSM**	Решение матричных уравнений

Список функций для работы с числами

Ниже перечислены математические функции:

Функция	Описание
a64l, **l64a**	Преобразование длинного целого в 64-разрядную текстовую строку и обратно
abs, **div**, **labs**, **ldiv**, **imul_dbl**, **umul_dbl, llabs, lldiv**	Модуль, частное и произведение двух целых чисел
asin, asinl, acos, acosl, atan, atanl, atan2, atan2l	Обратные тригонометрические функции
asinh, **acosh**, **atanh**	Обратные гиперболические функции
atof, **atoff**, **strtod**, **strtold**, **strtof**	Преобразование текстовой строки в число с плавающей точкой и обратно
bessell: j0, **j1**, **jn**, **y0**, **y1**, **yn**	Функции Бесселя
class, **finite**, **isnan**, **unordered**	Определение типов функций с плавающей точкой
copysign, **nextafter**, **scalb**, **logb**, **ilogb**	Некоторые бинарные функции с плавающей точкой
nrand48, **mrand48**, **jrand48**, **srand48**, **seed48**, **lcong48**	Генерация псевдослучайных последовательностей
lrand48_r, **mrand48_r**, **nrand48_r**, **seed48_r** и **srand48_r**	Генерация псевдослучайных последовательностей
drem и **remainder**	Остаток IEEE
ecvt, **fcvt**, **gcvt**	Преобразование числа с плавающей точкой в строку
erf, **erfl**, **erfc**, **erfcl**	Функция ошибок и дополнительная функция ошибок
exp, **expl**, **expm1**, **log**, **logl**, **log10**, **log10l**, **log1p**, **pow**, **powl**	Экспоненциальная, логарифмическая и степенная функции
floor, **floorl**, **ceil**, **ceill**, **nearest**, **trunc**, **rint**, **itrunc**, **uitrunc**, **fmod**, **fmodl**, **fabs**, **fabsl**	Округление числа с плавающей точкой
fp_any_enable, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, **fp_disable**	Разрешают выполнять операции при возникновении исключительной ситуации с плавающей точкой
fp_clr_flag, **fp_set_flag**, **fp_read_flag** или **fp_swap_flag**	Разрешают выполнять операции при возникновении исключительной ситуации с плавающей точкой
fp_invalid_op, **fp_divbyzero**, **fp_overflow**, **fp_underflow**, **fp_inexact**, **fp_any_xcp**	Обнаружение исключительной ситуации с плавающей точкой
fp_iop_snan, **fp_iop_infsinf**, **fp_iop_infdinf**, **fp_iop_zrdzr**, **fp_iop_infmzr**, **fp_iop_invcmp**	Обнаружение исключительной ситуации с плавающей точкой
fp_read_rnd,**fp_swap_rnd**	Чтение с округлением и включение режима округления IEEE
frexp, **frexpl**, **ldexp**, **ldexpl**, **modf**, **modfl**	Операции над числами с плавающей точкой
l64a_r	Преобразование 64-разрядного целого в строку
lgamma, **lgammal**, **gamma**	Логарифм гамма-функции
hypot, **cabs**	Функции расстояния в евклидовой метрике и модуля числа
13tol, **ltol3**	Преобразование трехбайтового целого в длинное целое и обратно
madd, **msub**, **mult**, **mdiv**, **pow**, **gcd**, **invert**, **rpow**, **msqrt**, **mcmp**, **move**, **min**, **omin**, **fmin**, **m_in**, **mout**, **omout**, **fmout**, **m_out**, **sdiv**, **itom**	Арифметические операции над целыми числами с большой точностью
rand, **srand**	Генерация случайных чисел
rand_r	Генерация случайных чисел
random, **srandom**, **initstate**, **setstate**	Генерация случайных чисел на основе улучшенного алгоритма
rsqrt	Вычисление величины, обратной квадратному корню из числа
sin, **cos**, **tan**	Тригонометрические и обратные тригонометрические функции
sinh, **sinhl**, **cosh**, **coshl**, **tanh**, **tanhl**	Гиперболические функции
sqrt, **sqrtl**, **cbrt**	Квадратный и кубический корень
strtol, **strtoll**, **strtoul**, **strtoull**, **atol**, **atoi**	Преобразование строки в целое

Список функций для работы с числами двойной длины

Ниже перечислены функции, предназначенные для работы с целыми числами двойной длины:

Функция	Описание
llabs	Вычисляет модуль целого числа двойной длины
lldiv	Вычисляет частное и остаток от деления двух целых чисел двойной длины
strtoll	Преобразует строку в целое число двойной длины со знаком
strtoull	Преобразует строку в целое число двойной длины без знака
wcstoll	Преобразует строку "широких" символов в целое число двойной длины со знаком
wcstoull	Преобразует строку "широких" символов в целое число двойной длины без знака

Список функций для работы с 128-разрядными числами двойной точности

Ниже перечислены функции, предназначенные для работы со 128-разрядными числами с плавающей точкой типа long double.

Эти функции не поддерживают 64-разрядные числа типа long double. В приложениях, работающих с 64-разрядным числами типа long double, нужно применять соответствующие функции с двойной точностью.

Функция	Описание
acosl	Вычисляет арксинус числа с плавающей точкой типа long double
asinl	Вычисляет арксинус числа с плавающей точкой типа long double
atan2l	Значение арктангенса *x*/*y* (x и y относятся к типу long double)
atanl	Арктангенс числа с плавающей точкой типа long double
ceill	Наименьшее целое, ближайшее (сверху) к указанному числу с плавающей точкой типа long double
coshl	Гиперболический косинус числа с плавающей точкой типа long double
cosl	Косинус числа с плавающей точкой типа long double
erfcl	Значение разности (1 - функция ошибок для числа с плавающей точкой типа long double)
erfl	Функция ошибок для числа с плавающей точкой типа long double
expl	Экспонента числа с плавающей точкой типа long double
fabsl	Модуль числа с плавающей точкой типа long double
floorl	Наибольшее целое, ближайшее (снизу) к указанному числу с плавающей точкой типа long double
fmodl	Остаток от деления *x*/*y* типа long double, где *x* и *y* - числа с плавающей точкой типа long double
frexpl	Выражает число с плавающей точкой типа long double в виде нормализованной мантиссы и целой степени двойки, сохраняет целое и возвращает мантиссу
ldexpl	Произведение числа с плавающей точкой типа long double на целую степень двойки
lgammal	Натуральный логарифм модуля гамма-функции для числа с плавающей точкой типа long double
log10l	Десятичный логарифм числа с плавающей точкой типа long double
logl	Натуральный логарифм числа с плавающей точкой типа long double
modfl	Сохраняет целую часть числа типа long double и возвращает дробную часть
powl	Значение *x* в степени *y*; x и y - числа с плавающей точкой типа long double
sinhl	Гиперболический синус числа с плавающей точкой типа long double
sinl	Синус числа с плавающей точкой типа long double
sqrtl	Квадратный корень из числа с плавающей точкой типа long double
strtold	Преобразует строку в число с плавающей точкой типа long double
tanl	Тангенс числа с плавающей точкой типа long double
tanhl	Гиперболический тангенс числа с плавающей точкой типа long double

Список функций для работы с процессами

С появлением поддержки нитей были добавлены новые и расширены некоторые старые функции для работы с процессами. Теперь планировщик управляет работой нитей, а не процессов.

Обработчик сигналов по-прежнему описывается на уровне процессов, однако для каждой нити можно задать маску сигналов. Вот несколько примеров измененных и новых функций: **getprocs**, **getthrds**, **ptrace**, **getpri**, **setpri**, **yield** и **sigprocmask**.

Эти функции можно отнести к следующим категориям:

Запуск процесса

exec: execl, execv, execle, execve, execlp, execvp или exect
Запуск в вызывающем процессе новой программы

fork или vfork
Создание нового процесса

reboot
Повторный запуск системы

siginterrupt
Задает список функций, которые должны запускаться повторно после получения специального сигнала прерывания

Остановка процесса

pause
Приостанавливает процесс до получения сигнала

wait, wait3, waitpid
Приостанавливает процесс до тех пор, пока не будет прерван или завершен дочерний процесс

Завершение процесса

abort
Завершает текущий процесс и создает дамп памяти путем отправки сигнала **SIGOT**

exit, atexit или _exit
Завершает процесс

, unatexit,
Аннулирует регистрацию процедур, которые были ранее зарегистрированы с помощью процедуры **atexit**. Если указанная функция будет найдена, она будет удалена из списка функций, вызываемых при нормальном завершении работы программы.

kill или killpg
Завершает текущий процесс или группу процессов путем отправки сигнала

Идентификация процессов и нитей

ctermid
Возвращает полное имя терминала, управляющего текущим процессом

cuserid
Возвращает буквенно-цифровое имя пользователя, связанное с текущим процессом

getpid, getpgrp или getppid
Возвращает ИД процесса, группы процессов и родительского процесса, соответственно

getprocs
Возвращает записи из таблицы процессов

getthrds
Возвращает записи из таблицы нитей

setpgid или setpgrp
Задает ИД группы процессов

setsid
Создает сеанс и задает идентификаторы групп процессов

uname или unamex
Возвращает имя текущей операционной системы

Учет ресурсов процесса

acct
Включает и выключает учет ресурсов процессов

ptrace
Трассировка процесса

Выделение ресурсов процессу

brk или sbrk
Изменяет размер сегмента данных

getdtablesize
Возвращает размер таблицы дескрипторов

getrlimit, setrlimit или vlimit
Устанавливают ограничения на системные ресурсы для текущего процесса

getrusage, times или vtimes
Выводит информацию об использовании ресурсов

plock
Блокируют процессы, текст или данные в памяти

profil
Запускает и завершает профайлер, собирающий информацию о частоте обращения по различным адресам программы

ulimit
Устанавливает ограничения на ресурсы для пользовательского процесса

Изменение приоритета процесса

getpri
Возвращает приоритет планирования процесса

getpriority, setpriority или nice
Возвращает или устанавливает значение приоритета процесса

setpri
Устанавливает постоянное значение приоритета планирования процесса

yield
Освобождает процессор для процессов с более высоким приоритетом

Синхронизация процессов и нитей

compare_and_swap
Обновляет переменную длиной в слово при выполнении заданного условия и возвращает ее значение; выполняется как атомарная операция

fetch_and_add
Обновляет переменную длиной в слово; выполняется как атомарная операция

fetch_and_and и fetch_and_or
Устанавливает или сбрасывает биты переменной длиной в слово; выполняется как атомарная операция

semctl
Управляет операциями над семафором

semget
Возвращает набор семафоров

semop
Выполняет операции над семафором

Работа с сигналами и масками

raise
Отправляет сигнал работающей программе

sigaction, sigvec или signal
Задает реакцию на получение сигнала

sigemptyset, sigfillset, sigaddset, sigdelset или sigismember
Предназначена для создания и работы с маской сигналов

sigpending
Задает набор сигналов, отправка которых блокируется

sigprocmask, sigsetmask или sigblock
Устанавливает маску сигналов

sigset, sighold, sigrelse или sigignore
Дополнительные функции обработки и управления сигналами

sigsetjmp или siglongjmp
Сохраняет и возвращает содержимое стека и маски сигналов

sigstack
Задает содержимое стека сигналов

sigsuspend
Изменяет набор заблокированных сигналов

ssignal или gsignal
Реализуют средство отправки сигналов

Работа с сообщениями

msgctl
Предназначена для управления сообщениями

msgget
Выводит на экран идентификатор очереди сообщений

msgrcv
Считывает сообщение из очереди

msgsnd
Отправляет сообщения в очередь сообщений

msgxrcv
Принимает расширенное сообщение

psignal
Печать сообщений о сигналах системы

Список функций для программирования с несколькими нитями

Для того чтобы обеспечить целостность данных, при создании программ с несколькими нитями следует применять реентерабельные функции.

Ниже приведен список реентерабельных функций, которые следует применять вместо их обычных аналогов:

Функция	Описание
asctime_r	Преобразует значение времени в массив символов
getgrnam_r	Возвращает следующую запись о группе с указанным именем из базы данных пользователей
getpwuid_r	Возвращает следующую запись из базы данных пользователей с указанным ИД

В следующем списке перечислены не реентерабельные функции из libc.

Функция	Описание		
asctime	getgrent	gsignal	setkey
auditread	getgrgid	hcreate	setlogmask
closelog	getgrnam	hdestroy	setnetent
crypt	getgroupsbyuser	hsearch	setnetgrent
ctime	getgrset	inet_ntoa	setprotoent
dirname	gethostbyaddr	initstate	setpwent
drand48	gethostbyname	innetgr	setpwfile
ecvt	gethostent	iso_addr	setrpcent
endttyent	getlogin	iso_ntoa	setservent
encrypt	getnetbyaddr	jrand48	setstate
asctime	getnetbyname	l64a	setttyent
endfsent	getnetent	lcong48	setutent
endfsent	getnetgrent	link_ntoa	setutxent
endgrent	getopt	localtime	srand48
endhostent	getprotobyname	lrand48	srandom
endnetent	getprotobynumber	mrand48	ssignal
endnetgrent	getprotoent	mtime	strerror
endprotoent	getpwent	ndutent	strtok
endpwent	getpwnam	nrand48	syslog
endrpcent	getpwuid	ns_ntoa	ttyname
endservent	getrpcbyname	openlog	utmpname
endttyent	getrpcbynumber	pututline	wcstok
endutxent	getrpcent	pututxline	
erand48	getservbyname	rand	
ether_aton	getservbyport	random	
ether_ntoa	getservent	rcmd	
fcvt fgetgrent	getttyent	rcmd2	
fgetpwent	getttynam	readdir	
getdate	getuinfo	rexec	
getfsent	getutent	re_comp	
getfsent	getutid	re_exec	
getfsfile	getutline	seed48	
getfsfile	getutxent	setfsent	
getfsspec	getutxid	setgrent	
getfstype	getutxline	sethostent	

Список функций библиотеки инструментальных средств программиста

Библиотека Programmers Workbench Library (**libPW.a**) поставляется для сохранения совместимости с уже существующими программами.

Применять эти функции в новых программах не рекомендуется. Эта библиотека соответствует стандарту AT&T PWB Toolchest.

Обычный	Описание
any (*символ, строка***)**	Определяет, содержится ли *символ* в *строке*
anystr (*строка1, строка2***)**	Определяет смещение в *строке1* первого символа, присутствующего также в *строке2*
balbrk (*строка, начало, конец, завершение***)**	Определяет смещение в *строке* первого символа из строки *завершение*, который не содержится в сбалансированной строке с параметрами *начало* и *конец*
cat (*приемник, источник1,источник0***)**	Объединяет строки *источник0* и *источник1* и копирует результат в *приемник*
clean_up ( )	Функция очистки по умолчанию
curdir (*строка***)**	Помещает в *строку* полное имя текущего каталога
dname (*p***)**	Определяет каталог, в котором находится *файл*
fatal (*сообщение***)**	Общая функция обработки ошибок
fdfopen (*дескриптор, режим***)**	Работает аналогично функции **stdio fdopen**
giveup (*дамп***)**	Принудительное создание дампа памяти
imatch (*подстрока, строка***)**	Определяет, начинается *строка* с *подстроки*
lockit (*файл_блокировки, число, идентификатор_процесса***)**	Создает файл блокировки
move (*строка1, строка2, n***)**	Копирует первые *n* символов из *строки1* в *строку2*
patoi (*строка***)**	Преобразует *строку* в тип int
patol (*строка***)**	Преобразует *строку* в тип long**.**
repeat (*приемник, строка, n***)**	Присваивает *приемнику* содержимое *строки*, повторенное *n* раз
repl (*строка, старый, новый***)**	Замещает каждое вхождение символа *старый* в *строке* символом *новый*
satoi (*строка, *переменная***)**	Преобразует *строку* в тип int записывает результат в **переменную*
setsig ( )	Включает обработку сигналов функцией **setsig1**
setsig1 (*сигнал***)**	Общая функция обработки сигналов
sname (*строка***)**	Возвращает указатель на имя элемента, входящего в полный путь *строка*
strend (*строка***)**	Ищет конец *строки.*
trnslat (*строка, исходные_символы, целевые_символы, приемник***)**	Копирует *строку* в *приемник*, заменяя все *исходные_символы* на *целевые_символы*
unlockit (*файл_блокировки, идентификатор_процесса***)**	Удаляет файл блокировки
userdir (*ИД-пользователя***)**	Возвращает имя начального каталога пользователя с указанным идентификатором
userexit (*код***)**	Пользовательская функция выхода по умолчанию
username (*ИД-пользователя***)**	Возвращает ИД пользователя
verify (*строка1, строка2***)**	Определяет смещение в *строке1* первого символа, который не совпадает со *строкой2*
xalloc (*asize***)**	Выделяет память
xcreat (*имя, режим***)**	Создает файл
xfree (*aptr***)**	Освобождает память
xfreeall ( )	Освобождает всю память
xlink (*файл1, файл2***)**	Связывает файлы
xmsg (*файл, функция***)**	Вызывает функцию **fatal** с указанным сообщением об ошибке
xpipe (*t***)**	Создает канал
xunlink (*f***)**	Удаляет запись каталога
xwrite (*дескриптор, буфер, n***)**	Записывает *n* байт из *буфера* в файл, связанный с *дескриптором*.
zero (*адрес, n***)**	Обнуляет первые *n* байт по указанному *адресу*

Обычный	Описание
zeropad (*s*)	Заменяет начальные пробелы *строки* нулями

Файл

/usr/lib/libPW.a
Содержит функции, предназначенные для сохранения совместимости с существующими программами.

Список функций защиты и контроля

В этом разделе перечислены функции защиты и контроля.

Функции управления доступом

Функция	Описание
acl_chg и **acl_fchg**	Изменяют информацию, управляющую доступом к файлу
acl_get и **acl_fget**	Возвращает информацию, управляющую доступом к файлу
acl_put и **acl_fput**	Задает информацию, управляющую доступом к файлу
acl_set и **acl_fset**	Задает основные записи, управляющие доступом к файлу
aclx_convert	Преобразует информацию об управлении доступом из одного типа ACL в другой
aclx_get или **aclx_fget**	Возвращает информацию, управляющую доступом к файлу, если связанный ACL имеет тип AIXC
aclx_gettypeinfo	Возвращает параметры ACL для типа ACL
aclx_gettypes	Извлекает список типов ACL, поддерживаемых для файловой системы, связанной с предоставленным путем
aclx_print или **aclx_printStr**	Преобразует двоичную информацию об управлении доступом в недвоичный, читаемый формат
aclx_put или **aclx_fput**	Сохраняет информацию об управлении доступом для объекта файловой системы
aclx_scan или **aclx_scanStr**	Преобразует информацию об управлении доступом из недвоичного читаемого текстового формата в двоичные данные ACL в исходном формате для данного типа ACL
chacl и **fchac l**	Изменяет права доступа к файлу
chmod и **fchmod**	Изменяет права доступа к файлу
chown, **fchown**, **chownx** и **fchownx**	Изменяет принадлежность файла
frevoke	Аннулирует права доступа к файлу, предоставленные другому процессу
revoke	Аннулирует права доступа к файлу
statacl и **fstatacl**	Возвращает информацию, управляющую доступом к файлу

Функции контроля

Функция	Описание
audit	Включает и выключает контроль системы
auditbin	Задает файлы для хранения контрольных записей
auditevents	Возвращает или устанавливает состояние контроля за событиями системы
auditlog	Добавляет контрольную запись в контрольный файл
auditobj	Возвращает или устанавливает режим контроля за объектом данных системы
auditpack	Упаковывает и распаковывает двоичный файл с контрольными записями
auditproc	Возвращает или задает состояние контроля за процессом
auditread и **auditread_r**	Считывает контрольную запись
auditwrite	Заносит в файл контрольную запись

Функции идентификации

Функции идентификации позволяют сохранить в памяти обычный и зашифрованный пароль. Учтите, что при этом можно будет узнать пароли, создав дамп памяти.

Функция	Описание
authenticate	Идентифицирует имя и пароль пользователя
ckuseracct	Проверяет правильность учетной записи пользователя
ckuserID	Идентифицирует пользователя
crypt, **encrypt** и **setkey**	Зашифровывает и расшифровывает данные
genpagvalue	Создает глобальное уникальное значение PAG для указанного имени PAG, например afs.
getpagvalue64	Выдает 64-разрядные значения PAG для процесса.
setpagvalue64	Сохраняет 64-разрядные значения PAG для процесса. .
getgrent, **getgrgid**, **getgrnam**, **setgrent** и **endgrent**	Предназначены для работы с базовой информацией о группе в базе данных пользователей
getgrgid_r	Получает ИД группы из записи базы данных группы в среде с несколькими нитями
getgrnam_r	Выполняет поиск имени в базе данных группы в среде с несколькими нитями
getgroupattr, **IDtogroup**, **nextgroup** и **putgroupattr**	Предназначены для работы с информацией о группе в базе данных пользователей
getlogin	Возвращает ИД пользователя
getlogin_r	Возвращает ИД пользователя в среде с несколькими нитями
getpass	Возвращает пароль
getportattr и **putportattr**	Предназначены для работы с информацией о порте из базы данных портов
getpwent, **getpwuid**, **getpwnam**, **putpwent**, **setpwent** и **endpwent**	Предназначены для работы с базовой информацией о пользователе из базы данных пользователей
getuinfo	Возвращает значение, связанное с пользователем
getuserattr, **IDtouser**, **nextuser** или **putuserattr**	Предназначены для работы с информацией о пользователей из базы данных пользователей
getuserpw, putuserpw или **putuserpwhist**	Предназначены для работы с идентификационными данными пользователя
loginfailed	Записывает информацию о неудачных попытках входа в систему
loginrestrictions	Проверяет, есть ли у пользователя права доступа к системе
loginsuccess	Записывает информацию о случаях успешного входа в систему
newpass	Создает новый пароль пользователя
passwdexpired	Проверяет, не истек ли срок действия пользовательского пароля
setpwdb и **endpwdb**	Открывает или закрывает базу данных идентификации
setuserdb и **enduserdb**	Открывает или закрывает базу данных пользователей
system	Выполняет команду оболочки
tcb	Изменяет состояние TCB файла

Функции для работы с процессами

Функция	Описание
getgid и **getegid**	Возвращает фактический ИД процесса и ИД группы процесса
getgroups	Возвращает информацию о группах, к которым относится текущий процесс
getpcred	Возвращает информацию о параметрах защиты текущего процесса
getpenv	Возвращает информацию о среде текущего процесса
getuid и **geteuid**	Возвращает фактический или действующий ИД пользователя для текущего процесса
initgroups	Инициализирует дополнительный ИД группы текущего процесса
kleenup	Очищает среду выполнения процесса
setgid, **setrgid**, **setegid** и **setregid**	Задает идентификаторы групп вызывающего процесса
setgroups	Задает дополнительный ИД группы текущего процесса
setpcred	Устанавливает параметры защиты текущего процесса
setpenv	Задает параметры среды текущего процесса
setuid, **setruid**, **setuid** и **setreuid**	Устанавливает ИД пользователя для процесса
usrinfo	Возвращает и задает пользовательскую информацию о владельце текущего процесса

Список функций для работы со строками

Функции работы со строками предназначены для решения следующих задач:

Функции работы со строками предназначены для решения следующих задач:

- Определение позиции символа в строке
- Поиск последовательности символов в строке
- Копирование строки
- Объединение строк
- Сравнение строк
- Преобразование строки
- Вычисление размера строки

При компиляции программ, применяющих функции работы со строками, не нужно указывать специальный флаг. Кроме того, в такие программы не нужно включать особый заголовочный файл.

Для работы со строками предназначены следующие функции:

bcopy, bcmp, bzero, ffs
Поразрядные и побайтовые операции над строками

gets, fgets
Получает строку из потока

puts, fputs
Записывает строку в проток

compile, step, advance
Компилирует и подставляет шаблоны регулярных выражений

strlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok
Операции над строками

jcode
Преобразует строку в 8-разрядные коды для обработки.

varargs
Предназначена для работы со списком параметров переменной длины

Пример: программа для работы с символами

В этом разделе приведен пример программы управления символами.

```
/*
Эта программа предназначена для демонстрации применения функций
"классификации и преобразования символов". Для получения символов
применяются функции
getchar и putchar из библиотеки stdio
Задачи программы:
- Считать ввод из stdin
- Проверить, что все символы - печатаемые
  символы ascii
- Преобразовать все прописные символы строчные
- Удалить лишние пробелы
- Выдать статистику по типам символов
Эта программа демонстрирует применение следующих
функций:

- getchar
```

```
- putchar

- isascii (ctype)

- iscntrl (ctype)

- isspace (ctype)

- isalnum (ctype)

- isdigit (ctype)

- isalpha (ctype)

- isupper (ctype)

- islower (ctype)

- ispunct (ctype)

- tolower (conv)

- toascii ( conv)
*/
#include <stdio.h> /* Обязательный включаемый файл */
#include <ctype.h>  /* Функции для определения типа
символов */
/* Различные счетчики для статистики */
int asciicnt, printcnt, punctcnt, uppercnt, lowercnt,
digcnt, alnumcnt, cntrlcnt, spacecnt, totcnt, nonprntcnt, linecnt, tabcnt ;
main()
{
int ch ; /* Обрабатываемый символ */
char c , class_conv() ;
asciicnt=printcnt=punctcnt=uppercnt=lowercnt=digcnt==0;
cntrlcnt=spacecnt=totcnt=nonprntcnt=linecnt=tabcnt=0;
alnumcnt=0;
while ( (ch =getchar()) != EOF )
{
totcnt++;
c = class_conv(ch) ;
putchar(c);
}
printf("Введено %d\n строк", linecnt);
printf(" Распределение символов по категориям:\n");
printf(" TOTAL  ASCII  CNTRL   PUNCT   ALNUM  DIGITS UPPER
LOWER   SPACE   TABCNT\n");
printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n",totcnt,
asciicnt, cntrlcnt, punctcnt, alnumcnt, digcnt, uppercnt,lowercnt, spacecnt, tabcnt );
}
char class_conv(ch)
char ch;
{
if (isascii(ch)) {
asciicnt++;
if ( iscntrl(ch) && ! isspace(ch)) {
```

```
nonprntcnt++ ;
cntrlcnt++ ;
return(' ');
}
else if ( isalnum(ch)) {
alnumcnt++;
if (isdigit(ch)){
digcnt++;
return(ch);
}
else if (isalpha(ch)){
if ( isupper(ch) ){
uppercnt++ ;
return(tolower(ch));
}
else if ( islower(ch) ){
lowercnt++;
return(ch);
}
else {
/*
Невозможная ситуация - символ алфавита должен быть либо
строчным, либо
прописным.
*/
fprintf(stderr,"Ошибка классификации %c \n",ch);
return(NULL);
}
}
else if (ispunct(ch) ){
punctcnt++;
return(ch);
}
else if ( isspace(ch) ){
spacecnt++;
if ( ch == '\n' ){
linecnt++;
return(ch);
}
while ( (ch == '\t' ) || ( ch == ' ' ) ) {
if ( ch == '\t' ) tabcnt ++ ;
else if ( ch == ' ' ) spacecnt++ ;
totcnt++;
ch = getchar();
}
ungetc(ch,stdin);
totcnt--;
return(' ');
}
else {
/*
Невозможная ситуация - любой символ ASCII
принадлежит одному из классов.
*/
fprintf(stderr,"Ошибка классификации %c \n",ch);
return (NULL);
}
```

```
}
else
{
fprintf(stdout,"Обнаружен не-ASCII символ\n");
return(toascii(ch));
}
}
```

Пример: программа поиска и сортировки

В этом разделе приведен пример программы поиска и сортировки.

```
/**В этой программе демонстрируется работа со следующими функциями:
Функция -qsort (библиотечная функция быстрой сортировки)
Функция -bsearch (библиотечная функция двоичного поиска)
Функции -fgets, fopen, fprintf, malloc, sscanf и strcmp.
Программа считывает записи двух файлов ввода в виде
строк и выводит на принтер или экран:
записи из file2, исключенные из file1
записи из file1, исключенные из file2
Программа считывает записи обоих файлов
и размещает их в двух массивах, которые сортируются
общем порядке с помощью функции qsort. Каждый элемент
выполняет поиск очередного вхождения элемента одного из массивов
в другом массиве. Если элемент не будет найден,
отправляется сообщение о том, что запись
не найдена. Такая процедура выполняется и для элементов
другого массива. В результате создается второй список исключений.
**/
#include <stdio.h>      /* файл библиотеки для
                         /*стандартных функции ввода-вывода*/
#include <search.h>  /*включаемый файл для qsort*/
#include <sys/errno.h>  /*включаемый файл для интерпретации
                        /*стандартных ошибок*/
#define MAXRECS 10000           /*ограничение на размер массива*/
#define MAXSTR   256             /*максимальная длина строки ввода*/
#define input1 "file1"  /*входной файл*/
#define input2 "file2"           /*второй файл ввода*/
#define out1   "o_file1"  /*выходной файл 1*/
#define out2   "o_file2"  /*выходной файл 2*/
main()
{
char *arr1[MAXRECS] , *arr2[MAXRECS] ;/*массивы для сохранения
входных записей*/
unsigned int num1 , num2;
/*отслеживание числа
                                /*входных записей. Unsigned int
                                /*гарантирует
                                /*совместимость
                                /*с библиотечной процедурой qsort.*/
int i ;
int compar();            /*функция для qsort и
                         /* bsearch*/
extern int errno ; /* регистрация всех сбоев системы */
FILE *ifp1 , *ifp2, *ofp1, *ofp2; /*указатели файлов для
ввода и вывода */
```

```
void *bsearch() ;
/*библиотечная процедура двоичного поиска*/
void qsort();           /*библиотечная процедура быстрой сортировки*/
char*malloc() ;         /*функция для выделения памяти*/
void exit() ;
num1 = num2 = 0;
/**Открытие входных и выходных файлов для чтения и записи
**/
if ( (ifp1 = fopen( input1 , "r" ) ) == NULL )
{
(void) fprintf(stderr,"Не удалось открыть %s\n",input1);
exit(-1);
}
if (( ifp2 = fopen( input2 , "r" )) == NULL )
{
(void) fprintf(stderr,"Не удалось открыть %s\n",input2);
exit(-1);
}
if (( ofp1 = fopen(out1,"w" )) == NULL )
{
(void) fprintf(stderr,"Не удалось открыть %s\n",out1);
exit(-1);
}
if (( ofp2 = fopen(out2,"w")) == NULL )
{
(void) fprintf(stderr,"Не удалось открыть %s\n", out2);
exit(-1);
}
/**Заполнение массивов данными из входных файлов. Readline
возвращает число записей ввода.**/
if ( (i = readline( arr1 , ifp1 ))  < 0 )
{
(void) fprintf(stderr,"Нет данных в %s. Выход\n",input1);
exit(-1);
}
num1 = (unsigned) i;
if ( (i = readline ( arr2 , ifp2)) < 0 )
{
(void) fprintf(stderr,"Нет данных в %s.  Выход\n",input2);
exit(-1);
}
num2 = (unsigned) i;
/**
Теперь массивы можно отсортировать с помощью функции qsort
**/
qsort( (char *)arr1 , num1 ,  sizeof (char *)  , compar);
qsort( (char *)arr2 , num2 ,  sizeof (char *)  , compar);
/**В ходе сортировки двух массивов в общем порядке
программа создает список элементов, содержащихся только в одном из массивов,
с помощью bsearch.
Проверяет все элементы array1 на вхождение в array2
**/
for ( i= 0 ; i < num1 ; i++ )
{
if ( bsearch((void *)&arr1[i] , (char *)arr2,num2,
sizeof(char *) , compar) == NULL )
```

```
{
(void)  fprintf(ofp1,"%s",arr1[i]);
}
} /**Один список исключений**/
/** Проверяет все элементы array2 на вхождение в array1**/
for ( i = 0 ; i < num2 ; i++ )
{
  if ( bsearch((void *)&arr2[i], (char *)arr1, num1
, sizeof(char *) , compar) == NULL )
{
(void) fprintf(ofp2,"%s",arr2[i]);
}
}
/**Задача выполнена, выход из функции**/
return(0);
}
/**Функция
считывает записи из входного файлв и заполняет два массива.**/
readline ( char **aptr, FILE *fp )
{
char str[MAXSTR] , *p ;
int i=0 ;
/**Построчное чтение входного файла**/
while ( fgets(str , sizeof(str) , fp ))
{
/**Выделение памяти. Если память не будет выделена -
fails, exit.**/
if ( (p = (char *)malloc ( sizeof(str))) == NULL )
{
(void) fprintf(stderr,"Недостаточно памяти\n");
return(-1);
}
else
{
if ( 0 > strcpy(p, str))
{
(void) fprintf(stderr,"Сбой strcpy \n");
return(-1);
}
i++ ; /* увеличение счетчика записей */
}
} /**Достигнут конец входного файла **/
return(i);/*возвращает число прочитанных записей*/
}
/**Нужно отсортировать массивы по значению
первого поля записей. Первое поле выделяется с помощью SSCANF**/
compar( char **s1 , char **s2 )
{
char st1[100] , st2[100] ;
(void) sscanf(*s1,"%s" , st1) ;
(void) sscanf(*s2,"%s" , st2) ;
/**Возвращает результат сравнения строк во внешнюю процедуру**/
return(strcmp(st1 , st2));
}
```

Список библиотек операционной системы

В этом разделе перечислены библиотеки операционной системы.

Библиотека	Описание
/usr/lib/libbsd.a	Библиотека Berkeley
/lib/profiled/libbsd.a	Оптимизированная библиотека Berkeley
/usr/ccs/lib/libcurses.a	Библиотека Curses
/usr/ccs/lib/libc.a	Библиотека стандартного ввода-вывода (stdio), стандартная библиотека (stdlib) языка C
/lib/profiled/libc.a	Библиотека стандартного ввода-вывода (stdio), оптимизированная стандартная библиотека языка C
/usr/ccs/lib/libdbm.a	Библиотека управления базами данных
/usr/ccs/lib/libi18n.a	Библиотека преобразования (layout)
/usr/lib/liblvm.a	Библиотека Администратора логических томов (LVM)
/usr/ccs/lib/libm.a	Библиотека математических функций (math)
/usr/ccs/lib/libp/libm.a	Оптимизированная библиотека математических функций (math)
/usr/lib/libodm.a	Библиотека Администратора объектных данных (ODM)
/usr/lib/libPW.a	Библиотека Programmers Workbench
/usr/lib/libpthreads.a	Библиотека нитей, соответствующая стандарту POSIX
/usr/lib/libqb.a	Библиотека работы с очередями
/usr/lib/librpcsvc.a	Библиотека поддержки Вызова удаленных процедур (RPC)
/usr/lib/librts.a	Библиотека служб времени выполнения
/usr/lib/libs.a	Функции защиты
/usr/lib/libsm.a	Библиотека управления системой
/usr/lib/libsrc.a	Библиотека Контроллера системных ресурсов (SRC)
/usr/lib/libmsaa.a	Математическая библиотека SVID (System V Interface Definition)
/usr/ccs/lib/libp/libmsaa.a	Оптимизированная математическая библиотека SVID (System V Interface Definition)
/usr/ccs/lib/libtermcap.a	Ввод-вывод на терминал
/usr/lib/liby.a	Библиотека YP (Желтые страницы)
/usr/lib/lib300.a	Функции работы с графикой для рабочих станций DASI 300
/usr/lib/lib300s.a	Функции работы с графикой для рабочих станций DASI 300s
/usr/lib/lib300S.a	Функции работы с графикой для рабочих станций DASI 300S
/usr/lib/lib4014.a	Функции работы с графикой для рабочих станций Tektronix 4014
/usr/lib/lib450.a	Функции работы с графикой для рабочих станций DASI 450
/usr/lib/libcsys.a	Службы расширений ядра
/usr/ccs/lib/libdbx.a	Библиотека отладки
/usr/lib/libgsl.a	Библиотека работы с графикой
/usr/lib/libieee.a	Библиотека операций с плавающей точкой по стандарту IEEE
/usr/lib/libIM.a	Библиотека работы с файлом настройки
/usr/ccs/lib/libl.a	Библиотека lex
/usr/lib/libogsl.a	Старая версия библиотеки работы с графикой
/usr/lib/liboldX.a	Библиотека X10
/usr/lib/libplot.a	Функции работы с графопостроителем
/usr/lib/librpcsvc.a	Службы RPC
/usr/lib/librs2.a	Аппаратно-зависимые функции **sqrt** и **itrunc**
/usr/lib/libxgsl.a	Функции работы с графикой в Enhanced X-Windows
/usr/lib/libX11.a	Базовая библиотека X11
/usr/lib/libXt.a	Библиотека X11 toolkit
/usr/lib/liby.a	Базовая библиотека yacc

System Management Interface Tool (SMIT)

Инструмент управления системой (SMIT) - это интерактивный расширяемый интерфейс, предназначенный для запуска команд.

Он представляет собой набор приглашений для ввода информации, необходимой для формирования текста команды, включая варианты предопределенных значений или динамические значения, используемые по

умолчанию. Тем самым при создании команд пользователь может сэкономить время и избежать значительной части ошибок, обычно допускаемых при вводе сложных команд, значений параметров, при записи системных команд или задании полных имен пользовательских оболочек.

Помимо изменения системной базы данных SMIT, устанавливаемой по умолчанию, вы можете создавать альтернативные базы данных и работать с ними.

Подробные сведения о программе SMIT приведены в следующих разделах:

В SMIT можно добавлять новые задачи, состоящие из одной или нескольких команд или сценариев оболочки **ksh**. Для этого необходимо добавить в базу данных SMIT новые экземпляры предопределенных объектов меню. Объекты меню (описываемые файлами настройки) применяются Администратором объектных данных (ODM) для обновления базы данных SMIT. Эта база данных управляет работой SMIT.

Информация, связанная с данной:

dspmsg

gencat

ksh

man

odmadd

odmcreate

odmget

smit

spaths

Типы окон SMIT

В Инструменте управления системой (SMIT) есть три типа окон. Окна образуют иерархию, в которую входят меню, списки вариантов и окна диалогов.

При выполнении задачи пользователь обычно последовательно открывает одно или несколько меню, затем несколько списков вариантов и, наконец, одно окно диалога.

В приведенной ниже таблице перечислены типы окон SMIT, описано содержимое окна и указано, что в каждом из них делает программа SMIT:

Тип окна	Что показано в окне	Какие действия выполняет SMIT
Меню	Список вариантов	Открывает новое окно в соответствии с выбранными значениями.
Список вариантов	Список вариантов или поле ввода	Получает данные для других окон. Дополнительно выбирает альтернативные окна диалогов или списки вариантов.
Окно диалога	Последовательность полей ввода.	Использует данные из полей ввода для составления текста команды и запуска соответствующей задачи.

Меню содержит список альтернативных подзадач; при выборе варианта из списка может появиться окно с другим меню или списком вариантов, либо окно диалога. Список вариантов обычно представляет собой набор элементов, содержащих информацию, необходимую для открытия следующего окна, либо перечень других списков вариантов или окон диалога, одно из которых можно открыть. В окне диалога пользователь вводит недостающие данные, и из него же запускается выбранная задача.

Меню представляет собой основную точку входа в программу SMIT и может вызывать другое меню, список вариантов или окно диалога. Из списка вариантов можно вызывать окно диалога. И, наконец, окно диалога - это последняя панель ввода в последовательности окон SMIT.

Меню

Меню SMIT представляет собой список пунктов, из которых пользователь может выбирать нужный пункт. Пункты меню - это, как правило, задачи или классы задач, которые могут выполняться с помощью SMIT. Главное меню SMIT содержит набор пунктов, каждый из которых определяет широкий диапазон системных задач. При выборе пунктов меню следующих иерархических уровней круг задач все более сужается, пока, наконец, пользователь не попадет в последнее окно диалога, предназначенное для ввода информации, требующейся для выполнения конкретной задачи.

Создавайте меню для пользователей SMIT таким образом, чтобы каждый пункт в нем соответствовал конкретной задаче. Вы можете создавать как простые меню и окна диалога, добавляя их к существующей ветви SMIT, так и сложные конструкции, представляющие собой иерархию меню, списков вариантов и окон выбора.

Во время работы программа SMIT получает все необходимые объекты меню с заданным идентификатором (значение дескриптора **id**) из хранилища объектов. Для того чтобы добавить пункт в какое-либо меню SMIT, необходимо добавить объект меню с идентификатором, равным значению дескриптора **id** других объектов этого меню.

Для создания меню определите его в файле настройки, а затем обработайте файл командой **odmadd**. В результате компиляции определение меню будет помещено в группу объектов меню. В одном или нескольких файлах можно определять любое количество меню, списков вариантов и окон диалога.

Команда	Описание
odmadd	Добавляет определения меню в хранилище объектов.
/usr/lib/objrepos	Хранилище объектов, применяемое по умолчанию для записи системной информации; вы можете использовать его для хранения откомпилированных объектов.

При работе программы SMIT объекты автоматически восстанавливаются из базы данных SMIT.

Примечание: Перед тем как удалять или добавлять какие-либо объекты или классы объектов, необходимо создать резервную копию каталога **/usr/lib/objrepos**. Случайное повреждение объектов или классов, необходимых для выполнения системных операций, может вызвать сбой в системе.

Списки

Список вариантов SMIT предлагает пользователю выбрать какой-либо элемент (как правило, системный объект - принтер, дисплей и т.д.) или атрибут объекта (например, последовательный или параллельный режим подключения принтера). Эта информация обычно используется программой SMIT для открытия следующего окна диалога.

Например, пользователь может выбрать из списка вариантов имя логического тома, характеристики которого он будет изменять. Это имя может использоваться как параметр в поле `sm_cmd_hdr.cmd_to_discover_postfix` вызываемого окна диалога. Оно может также использоваться в качестве значения поля `sm_cmd_opt.cmd_to_list_postfix`, или непосредственно - в качестве начального значения поля ввода открываемого окна диалога. В любом случае, для сохранения логической непротиворечивости необходимо, чтобы значение поля было определено до того, как будет открыто окно диалога, и оставалось постоянным все время, пока окно диалога открыто.

Создайте окно списка вариантов, в котором у пользователя будет запрашиваться один элемент данных (одно значение). Список вариантов в иерархии окон занимает промежуточное место между меню и окнами диалогов. Списки можно связывать, и создавать наборы, позволяющие получать значения сразу нескольких полей (объектов) в открываемом окне диалога.

Окно списка вариантов обычно состоит из приглашения (на естественном языке) и поля, предназначенного для ввода информации, либо всплывающего списка, из которого пользователь может выбрать значение; т.е.

одного поля запроса и одного поля ответа. Обычно показывается приглашение, а пользователь SMIT вводит нужное значение в поле ответа либо с помощью клавиатуры, либо выбирая его из списка вариантов или кольцевого списка опций.

Для создания динамического списка вариантов с ним можно связать команду (определяемую в поле `sm_cmd_opt.cmd_to_list`), которая будет создавать список возможных вариантов в момент открытия окна. В этом случае значения в списке не определяются на стадии программирования файла настройки, а создаются указанной командой на основе данных стандартного вывода. Для получения этого списка в интерфейсе SMIT нужно нажать клавишу **F4 (Esc+4)=Список**.

Если в поле `sm_cmd_opt.cmd_to_list` для динамического списка вариантов (`sm_cmd_hdr.ghost="y"`) определена команда, то она запускается автоматически. Окно списка вариантов не показывается, а пользователь видит только всплывающий список.

Применение расширенного динамического списка вариантов позволяет организовать переключение между списками вариантов. При этом список вариантов, из которого пользователь выбирает открываемое меню, определяется состоянием системы, а не данными, введенными пользователем. В этом случае для получения необходимой информации и выбора открываемого окна можно использовать дескриптор **cmd_to_classify** в расширенном динамическом списке вариантов.

Для создания списка вариантов определите его в файле настройки, а затем обработайте этот файл командой **odmadd**. В одном файле можно определять несколько меню, списков вариантов и окон диалогов. Команда **odmadd** добавляет список вариантов в определенное хранилище объектов. По умолчанию для хранения системной информации предназначен каталог **/usr/lib/objrepos**; вы можете использовать его в качестве хранилища откомпилированных объектов. При работе программы SMIT объекты автоматически восстанавливаются из базы данных SMIT.

Примечание: Перед тем как удалять или добавлять какие-либо объекты или классы объектов, необходимо создать резервную копию каталога **/usr/lib/objrepos**. Случайное повреждение объектов или классов, необходимых для выполнения системных операций, может вызвать сбой в системе.

Окна диалога

Окно диалога в SMIT - это интерфейс, с помощью которого пользователь может выполнять команды или задачи. В одном окне диалога выполняется одна или несколько команд, функций оболочки и т.п. Для одной команды можно создавать любое число окон диалога.

Для создания окна диалога необходимо определить, какую часть текста команды собираетесь создавать вы сами, а какие опции и операнды будут вводиться пользователем. В окне диалога для каждой из этих опций или операндов необходимо предусмотреть поле приглашения (на естественном языке) и поле ответа, предназначенное для ввода информации пользователем. Каждой опции и операнду соответствует объект опции команды в базе данных Администратора объектных данных (ODM). Полностью окно диалога определяется объектом заголовка окна диалога.

Пользователь SMIT вводит значение в поле ответа либо с помощью клавиатуры, либо выбирая его из списка вариантов или кольцевого списка опций. Для динамического формирования списка вариантов с объектом окна диалога можно связать команду, создающую список возможных вариантов. Соответствующие команды указываются в поле `sm_cmd_opt.cmd_to_list`. Для получения этого списка в интерфейсе SMIT нужно нажать клавишу **F4 (Esc + 4)=Список**. При нажатии этой клавиши программа SMIT выполняет команду, которая задана в поле `cmd_to_list`, а затем создает требуемый список на основании вывода этой команды и содержимого файла **stderr**.

С записями в окнах диалога могут быть связаны следующие опции:

Опция	Функция
#	Указывает, что ожидается ввод числового значения.
*	Обозначает обязательную запись.
+	Указывает, что с помощью клавиши F4 можно просмотреть список возможных значений.

В случае динамического окна диалога само окно на экране не появляется. Команда (задача) этого окна диалога запускается на выполнение, как если бы клавиша **Enter** в нем уже была нажата пользователем.

Для создания окна диалога определите его в файле настройки и обработайте командой **odmadd**. В одном файле можно определять несколько меню, списков вариантов и окон диалогов. Команда **odmadd** добавляет окно диалога в определенное хранилище объектов. По умолчанию для хранения системной информации предназначен каталог **/usr/lib/objrepos**; вы можете использовать его в качестве хранилища откомпилированных объектов. При работе программы SMIT объекты автоматически восстанавливаются из базы данных SMIT.

Примечание: Перед тем как удалять или добавлять какие-либо объекты или классы объектов, необходимо создать резервную копию каталога **/usr/lib/objrepos**. Случайное повреждение объектов или классов, необходимых для выполнения системных операций, может вызвать сбой в системе.

Классы объектов SMIT

Класс объектов Инструмента управления системой (SMIT), создаваемый с помощью Администратора объектных данных (ODM), определяет общий формат или тип данных для всех объектов, представляющих собой экземпляры данного объектного класса.

Следовательно, объектный класс SMIT - это, в основном, тип данных, а объект SMIT - конкретная запись этого типа.

Меню, списки вариантов и окна диалогов SMIT описываются объектами, представляющими собой экземпляры одного из четырех объектных классов:

- **sm_menu_opt**
- **sm_name_hdr**
- **sm_cmd_hdr**
- **sm_cmd_opt**

В приведенной ниже таблице перечислены объекты, применяемые для создания различных типов окон:

Тип окна	Класс объектов	Использование объекта (типичный случай)
Меню	**sm_menu_opt**	1 для заголовка окна
	sm_menu_opt	1 для первого элемента
	sm_menu_opt	1 для второго элемента
	...	...
	sm_menu_opt	1 для последнего элемента
Список вариантов	**sm_name_hdr**	1 для заголовка окна и других атрибутов
	sm_cmd_opt	1 для поля ввода и всплывающего списка
Окно диалога	**sm_cmd_hdr**	1 для заголовка окна и командной строки
	sm_cmd_opt	1 для первого поля ввода
	sm_cmd_opt	1 для второго поля ввода
	...	...
	sm_cmd_opt	1 для последнего поля ввода

Каждый объект состоит из последовательности именованных полей и связанных с ними значений. Эта последовательность задается в ASCII-файлах настройки, которые могут применяться командой **odmadd** для инициализации или расширения баз данных SMIT. Разделы в файле должны быть отделены друг от друга одной или несколькими пустыми строками.

> **Примечание:** Комментарии в файле ввода ODM (ASCII-файле настройки), используемом командой **odmadd**, должны находиться в строке, содержащей в первой позиции символ # (знак фунта) или * (звездочку). Комментарий, начинающийся со звездочки (*), может находиться в строке раздела описания, но должен стоять после значения дескриптора.

Ниже приведен пример раздела описания объекта **sm_menu_opt**:

```
sm_menu_opt:                    *имя класса объектов
  id             = "top_menu"   *имя объекта (окна меню)
  id_seq_num     = "050"
  next_id        = "commo"      *ИД объектов для следующего окна
  text           = "Приложения и службы связи"
  text_msg_file  = ""
  text_msg_set   = 0
  text_msg_id    = 0
  next_type      = "m"          *next_id для определения другого меню
  alias          = ""
  help_msg_id           = ""
  help_msg_loc          = ""
  help_msg_base = ""
  help_msg_book = ""
```

Для описания значений полей объекта обычно применяется запись в формате *класс-объектов.дескриптор*. Например, в предыдущем примере описания объекта **sm_menu_opt** значение **sm_menu_opt.id** равно `top_menu`.

Ниже приведен пример раздела описания объекта **sm_name_hdr**:

```
sm_name_hdr:              *---- применяется в списках вариантов
  id                      = ""  *имя данного окна списка вариантов
  next_id                 = ""  *следующий sm_name_hdr или sm_cmd_hdr
  option_id               = ""  *определяет один связанный sm_cmd_opt
  has_name_select        = ""
  name                    = "" *заголовок данного окна
  name_msg_file         = ""
  name_msg_id           = 0
  type                    = ""
  ghost                   = ""
  cmd_to_classify         = ""
  cmd_to_classify_postfix = ""
  raw_field_name          = ""
  cooked_field_name       = ""
  next_type               = ""
  help_msg_id           = ""
  help_msg_loc          = ""
  help_msg_base = ""
  help_msg_book = ""
```

Ниже приведен пример раздела описания объекта **sm_cmd_hdr**:

```
sm_cmd_hdr:                     *---- применяется в списках вариантов
  id                      = "" *имя данного окна диалога
  option_id               = "" *определяет связанный набор объектов sm_cmd_opt
  has_name_select        = ""
  name                    = "" *заголовок данного окна
  name_msg_file         = ""
  name_msg_set          = 0
  name_msg_id           = 0
  cmd_to_exec             = ""
  ask                     = ""
```

```
  exec_mode               = ""
  ghost                   = ""
  cmd_to_discover         = ""
  cmd_to_discover_postfix = ""
  name_size               = 0
  value_size              = 0
  help_msg_id           = ""
  help_msg_loc          = ""
  help_msg_base = ""
  help_msg_book = ""
```

Ниже приведен пример раздела описания объекта **sm_cmd_opt**:

```
sm_cmd_opt:              *---- применяется для списков вариантов и окон диалогов
  id                    = ""  *имя данного объекта
  id_seq_num            = ""  *"0" для окна списка вариантов
  disc_field_name       = ""
  name                  = ""  *описание данного поля
  name_msg_file         = ""
  name_msg_set          = 0
  name_msg_id           = 0
  op_type               = ""
  entry_type            = ""
  entry_size               = 0
  required                 = ""
  prefix                = ""
  cmd_to_list_mode      = ""
  cmd_to_list           = ""
  cmd_to_list_postfix   = ""
  multi_select          = ""
  value_index           = 0
  disp_values           = ""
  values_msg_file       = ""
  values_msg_set        = 0
  values_msg_id         = 0
  aix_values            = ""
  help_msg_id           = ""
  help_msg_loc          = ""
  help_msg_base = ""
  help_msg_book = ""
```

Во всех объектах SMIT предусмотрено поле `id`, содержащее имя, по которому можно найти данный объект. Для поиска объектов **sm_menu_opt**, представляющих заголовки меню, применяется поле `next_id`. Для объектов **sm_menu_opt** и **sm_name_hdr** задаются поля `next_id`, указывающие на поля `id` других объектов. Именно таким образом реализованы связи между окнами в базе данных SMIT. Аналогично, в объектах **sm_name_hdr** и **sm_cmd_hdr** есть поле `option_id`, которое указывает на поля `id` связанных с ними объектов **sm_cmd_opt**.

Примечание: Значение поля **sm_cmd_hdr.option_id** совпадает со значениями полей **sm_cmd_opt.id**; таким образом определяется связь между объектом **sm_cmd_hdr** и объектами **sm_cmd_opt**.

Один и тот же объект **sm_cmd_opt** может применяться несколькими окнами диалогов, поскольку для поиска объектов с одинаковыми значениями полей `sm_cmd_opt.id` SMIT использует оператор ODM **LIKE**. SMIT разрешает указывать в поле `sm_cmd_hdr.option_id` до пяти идентификаторов (разделенных запятыми), поэтому с объектом **sm_cmd_hdr** можно связывать объекты **sm_cmd_opt**, которым присвоено любое из пяти значений поля `sm_cmd_opt.id`.

Из приведенной ниже таблицы видно, каким образом значение поля `sm_cmd_hdr.option_id` связано со значениями полей `sm_cmd_opt.id` и `sm_cmd_opt.id_seq_num`.

> **Примечание:** Значения полей `sm_cmd_opt.id_seq_num` применяются для сортировки объектов, восстанавливаемых из хранилища для открываемого окна.

ИД объектов, которые должны быть получены (sm_cmd_hdr.option_id)	Полученные объекты (sm_cmd_opt.id)	Последовательность полученных объектов в окне (sm_cmd_opt.id_seq_num)
"demo.[AB]"	"demo.A"	"10"
	"demo.B"	"20"
	"demo.A"	"30"
	"demo.A	"40"
"demo.[ACD]"	"demo.A"	"10"
	"demo.C"	"20"
	"demo.A"	"30"
	"demo.A"	"40"
	"demo.D"	"50"
"demo.X,demo.Y,demo.Z"	"demo.Y"	"20"
	"demo.Z"	"40"
	"demo.X"	"60"
	"demo.X"	"80"

База данных SMIT

Объекты SMIT генерируются с помощью средств создания ODM и хранятся в файлах заданной базы данных. По умолчанию база данных SMIT состоит из восьми файлов:

- **sm_menu_opt**
- **sm_menu_opt.vc**
- **sm_name_hdr**
- **sm_name_hdr.vc**
- **sm_cmd_hdr**
- **sm_cmd_hdr.vc**
- **sm_cmd_opt**
- **sm_cmd_opt.vc**

По умолчанию для хранения файлов применяется каталог **/usr/lib/objrepos**. Файлы всегда должны сохраняться и восстанавливаться согласованно.

Псевдонимы и команды быстрого доступа SMIT

В этом разделе рассмотрены псевдонимы и команды быстрого доступа SMIT.

Для определения команд быстрого доступа в Инструменте управления системой (SMIT) предназначен объект **sm_menu_opt**. Если при запуске SMIT вместе с командой **smit** указать команду быстрого доступа, то вы сразу перейдете к нужному меню, списку вариантов или окну диалога. При этом сам псевдоним показан не будет. Для одного и того же меню, списка вариантов или окна диалога может существовать несколько команд быстрого доступа.

Для того чтобы определить команду быстрого доступа в объекте **sm_menu_opt**, укажите в поле `sm_menu_opt.alias` значение `"y"`. В этом случае объект **sm_menu_opt** будет применяться исключительно для задания команд быстрого доступа. Для задания команды быстрого доступа или псевдонима предназначено поле `sm_menu_opt.id`. Значение поля `sm_menu_opt.next_id` указывает на другой объект меню, объект заголовка списка вариантов или объект заголовка окна диалога (тип объекта зависит от значения поля `sm_menu_opt.next_type`: `"m"` - меню, `"n"` - селектор, `"d"` - окно).

Если в объекте **sm_menu_opt**, связанном с заголовком меню (`next_type="m"`), определяется не псевдоним, то значение в поле `sm_menu_opt.next_id` должно быть уникальным, поскольку оно автоматически используется как команда быстрого доступа.

Если вы хотите, чтобы два пункта меню ссылались на одно и то же меню следующего уровня, то одно из полей `next_id` должно ссылаться на псевдоним этого меню.

Для создания псевдонима и команды быстрого доступа определите их в файле настройки, а затем обработайте этот файл с помощью команды **odmadd**. В одном файле можно определять несколько меню, списков вариантов и окон диалогов. Команда **odmadd** добавляет определение псевдонима в определенное хранилище объектов. По умолчанию для хранения системной информации предназначен каталог **/usr/lib/objrepos**; вы можете использовать его в качестве хранилища откомпилированных объектов. При работе программы SMIT объекты автоматически восстанавливаются из базы данных SMIT.

> **Примечание:** Перед тем как удалять или добавлять какие-либо объекты или классы объектов, необходимо создать резервную копию каталога **/usr/lib/objrepos**. Случайное повреждение объектов или классов, необходимых для выполнения системных операций, может вызвать сбой в системе.

Дескрипторы информационных команд SMIT

Для получения информации, необходимой для продолжения работы с интерфейсом SMIT (например, текущих значений динамических параметров), в Инструменте управления системой (SMIT) предусмотрено несколько специальных дескрипторов.

Каждый из этих дескрипторов связан с текстом команды, применяемой для получения необходимых данных.

Для определения команд, получающих информацию, применяются следующие дескрипторы:

- Дескриптор **cmd_to_discover** класса объектов **sm_cmd_hdr**, применяемый для определения заголовка окна диалога.
- Дескриптор **cmd_to_classify** класса объектов **sm_name_hdr**, применяемый для определения заголовка списка вариантов.
- Дескриптор **cmd_to_list** класса объектов **sm_cmd_opt**, применяемый для задания списка вариантов или набора опций команды окна диалога, связанной с полем ввода.

При выполнении команды, определенной дескриптором **cmd_to_list**, **cmd_to_classify** или **cmd_to_discover**, SMIT сначала создает дочерний процесс. Стандартный вывод сообщений об ошибках (strerr) и стандартный вывод дочернего процесса перенаправляются в SMIT с помощью конвейера. Затем в дочернем процессе SMIT выполняет функцию **setenv("ENV=")**, чтобы исключить возможность автоматического запуска команд из пользовательского файла **$HOME/.env** при запуске новой оболочки. Затем SMIT вызывает функцию **execl**, которая запускает оболочку **ksh**, используя в качестве значения параметра строку **ksh -c**. Если при выходе возвращается значение 0, SMIT сообщает пользователю, что команда не выполнена.

SMIT задает полные имена файлов протокола и флаги **подробного вывода**, **трассировки** и **отладки**, применяемые в текущей среде выполнения команды. Указанные значения задаются с помощью следующих переменных среды:

- **_SMIT_LOG_FILE**
- **_SMIT_SCRIPT_FILE**
- **_SMIT_VERBOSE_FLAG**
- **_SMIT_TRACE_FLAG**
- **_SMIT_DEBUG_FLAG**

Наличие или отсутствие соответствующего флага обозначается значением 1 или 0.

Для того чтобы просмотреть текущие значения, вызовите функцию оболочки после запуска SMIT, а затем выполните команду **env | grep _SMIT**.

Операции записи в файлы протокола должны выполняться в режиме добавления и сопровождаться немедленной записью содержимого буферных файлов на диск (если это не происходит автоматически).

Дескриптор cmd_to_discover

При создании окна диалога программа SMIT получает из хранилища объектов заголовок окна диалога (объект **sm_cmd_hdr**) и тело окна диалога (один или несколько объектов **sm_cmd_opt**). Однако объекты **sm_cmd_opt** можно инициализировать и динамическими значениями. Если поле `sm_cmd_hdr.cmd_to_discover` не пустое (не `""`), то для получения текущих значений SMIT запускает команду, указанную в этом поле.

В качестве значения дескриптора **cmd_to_discover** можно задать любую командную строку **ksh**. Стандартный вывод команды должен иметь следующий формат:

```
#имя_1:имя_2: ... :имя_n\n
значение_1:значение_2: ... :значение_n
```

В стандартном выводе команды первым символом всегда должен быть знак фунта (`#`). Для отделения строки имен от строки значений необходимо указывать символ новой строки (`\n`). Если имен и значений несколько, они разделяются двоеточиями (`:`). Любое имя и значение может быть пустым (чему соответствуют два последовательных двоеточия без пробела между ними). SMIT поддерживает набор внутренних текущих значений в этом формате и использует его для передачи пар "имя-значение" из одного меню в другое.

> **Примечание:** Если значение содержит символ `:` (двоеточие), то перед `:` необходимо указать `#!` (знак фунта и восклицательный знак). В противном случае символ `:` будет восприниматься программой SMIT как разделитель полей.

Когда SMIT запускает команду, указанную в поле `cmd_to_discover`, он перехватывает данные из `стандартного вывода` команды и загружает указанные пары "имя-значение" (`имя_1` и `значение_1` `имя_2` и `значение_2` и т.д.) в дескрипторы **disp_values** и **aix_values** объектов **sm_cmd_opt** (опция команды окна диалога), сравнивая эти имена с дескрипторами **sm_cmd_opt.disc_field_name** объектов **sm_cmd_opt**.

В дескрипторе **disc_field_name** объекта **sm_cmd_opt** (опция команды окна диалога), представляющего выбранное ранее значение, должно быть указано имя `"_rawname"` или `"_cookedname"` (или любое другое имя, переопределяющее имя по умолчанию). В этом случае дескриптор **disc_field_name** диалогового окна **sm_cmd_opt** не будет полем ввода. Если команде всегда должно передаваться некоторое значение, то дескриптору **required** диалогового окна **sm_cmd_opt** нужно присвоить значение `y` (да) или одно из аналогичных значений.

В случае инициализации кольцевого списка опций в качестве значения по умолчанию, или начального значения поля ввода, в дескрипторе **cmd_to_discover** можно задать текущее значение (то есть, любую пару "имя-значение" из текущего набора значений окна). Если при инициализации окна диалога поле ввода окна совпадает с именем из текущего набора имен для этого окна (хранящегося в **sm_cmd_opt.disc_field_name**), то система проверят, не является ли поле кольцевым списком опций (**sm_cmd_opt.op_type = "r"**), и нет ли для него предопределенных значений кольцевого списка (**sm_cmd_opt.aix_values != ""**). Если это действительно поле кольцевого списка, то указанный набор значений кольцевого списка сравнивается с текущим значением **disc_field_name**. Если для какой-либо опции обнаруживается совпадение, то это значение из кольцевого списка опций становится значением по умолчанию (в **sm_cmd_opt.value_index** указывается его индекс) и выдается соответствующее преобразованное значение (**sm_cmd_opt.disp_values**), если оно есть. Если совпадения не обнаружено, то выдается сообщение об ошибке, и в качестве значения по умолчанию принимается текущее значение, которое становится единственным вариантом в кольцевом списке.

Во многих случаях команды получения информации уже существуют. Есть общие принципы работы с командами добавления, изменения, удаления и просмотра устройств и памяти. Например, пусть окну диалога команды **mk** ("добавить") необходимо получить значения определенных параметров. Для этого окно диалога может вызвать команду просмотра (**ls**) с параметром, для которого нужны значения по умолчанию. Для получения значений по умолчанию программа SMIT использует стандартный вывод команды просмотра (**ls**). Для некоторых объектов значения по умолчанию постоянны и известны уже на стадии разработки (т.е. не зависят от текущего состояния компьютера); в этом случае значения по умолчанию могут инициализироваться непосредственно в записи кода для окна диалога, а дескриптор **cmd_to_discover** не нужен. Затем окно диалога показывается на экране. После заполнения всех полей ввода в окне диалога и их подтверждения выполняется команда **mk**.

В качестве другого примера рассмотрим окно диалога команды **ch**. В нем для получения текущих значений параметров, например, конкретного устройства, применяется команда **ls**. Программа SMIT использует стандартный вывод этой команды (**ls**) для заполнения полей окна перед его выдачей на экран. В этом примере для получения информации применяется та же команда **ls**, что и в примере с командой (**mk**), за исключением некоторых небольших различий в наборе опций.

Дескрипторы cmd_to_*_postfix

Этот дескриптор определяет постфикс команды, заданной дескриптором **cmd_to_discover**, **cmd_to_classify** или **cmd_to_list**. Постфикс - это строка символов, определяющая флаги и параметры, которые добавляются к команде перед ее выполнением.

Для определения постфикса, добавляемого к команде, могут применяться следующие дескрипторы:

- **cmd_to_discover_postfix** - определяет постфикс дескриптора **cmd_to_discover** в объекте заголовка окна диалога **sm_cmd_hdr**.
- **cmd_to_classify_postfix** - определяет постфикс дескриптора **cmd_to_classify** в объекте заголовка списка вариантов **sm_name_hdr**.
- **cmd_to_list_postfix** - определяет постфикс дескриптора **cmd_to_list** в объекте **sm_cmd_opt**, задающем поле ввода, связанное со списком вариантов или окном диалога.

В приведенном ниже примере показано, как использовать постфиксные дескрипторы для определения флагов и значений параметров. Символ `*` (звездочка) в примере заменяет `list`, `classify` или `discover`.

Пусть для `cmd_to_*` задано значение `"DEMO -a"`, для `cmd_to_*_postfix` - значение `"-l _rawname -n stuff -R _cookedname"`, и указаны следующие текущие значения:

```
#имя1:_rawname:_cookedname::stuff\n
значение1:gigatronicundulator:parallel:xxx:47
```

Тогда будет сформирована команда следующего вида:

```
DEMO -a -l 'gigatronicundulator' -n '47' -R 'parallel'
```

Значения постфиксного дескриптора можно заключать в одинарные кавычки (`''`). В этом случае пробелы, содержащиеся в значениях параметров, будут обрабатываться правильно.

Создание и выполнение команд SMIT

Каждое окно диалога Инструмента управления системой (SMIT) формирует и запускает некоторую стандартную команду.

Команда, выполняемая с помощью окна диалога, определяется дескриптором **cmd_to_exec** в объекте **sm_cmd_hdr**, задающем заголовок окна диалога.

Создание задач, связанных с окном диалога

При создании команды, определенной в дескрипторе **sm_cmd_hdr.cmd_to_exec**, SMIT дважды просматривает набор объектов **sm_cmd_opt** окна диалога для получения префиксов и значений параметров. К ним относятся параметры, начальные значения которых были изменены пользователем, а также параметры с дескриптором **sm_cmd_opt.required**, равным "y".

При первом проходе собираются все значения объектов **sm_cmd_opt** (по порядку), для которых значение дескриптора **prefix** представляет собой пустую строку ("") или начинается со знака минус (-). Эти параметры добавляются к команде сразу после имени (в любом порядке) вместе с содержимым дескриптора **prefix**.

При втором проходе собираются все значения остальных объектов **sm_cmd_opt** (по порядку), для которых дескриптор **prefix** состоит из двух тире (-). Порядок следования этих параметров в команде имеет значение; они добавляются после собранных при первом проходе опций с флагами.

> **Примечание:** SMIT в точности исполняет то, что указано в поле префикса. Если в поле префикса указан символ, зарезервированный для команд оболочки, например, звездочка (*), то перед этим символом необходимо указать двойное тире и одинарную кавычку (—'). Если этого не сделать, то при анализе символа система может принять его за символ оболочки.

Если дескрипторы **disc_field_name** объектов **sm_cmd_opt** совпадают с именами параметров, созданных предыдущими списками вариантов или предыдущей командой получения информации, то значения параметров команды в окне диалога подставляются автоматически. Эти значения параметров считаются значениями по умолчанию и, как правило, в текст команды не добавляются. Если дескриптор **sm_cmd_opt.required** инициализирован значением "y" или "+", то соответствующие значения параметров будут добавляться в текст команды даже в том случае, если они не были изменены в диалоговом окне. Если значение дескриптора **sm_cmd_opt.required** равно "?", то соответствующие значения используются только в том случае, если связанные с ними поля ввода не пустые. Эти значения параметров встраиваются в текст команды в соответствии с описанной выше двухпроходной процедурой.

Если значение дескриптора **sm_cmd_opt.entry_type** не равно "r", то из значений параметров удаляются начальные и конечные пробелы и символы табуляции. Если полученное в результате значение параметра представляет собой пустую строку, других действий выполнено не будет, если только дескриптор **sm_cmd_opt.prefix** не указан с флагом опции. Если дескриптор **prefix** не равен "-", то значение параметра заключается в одинарные кавычки. Параметр помещается сразу за связанным с ним префиксом (если он существует), без пробела. Кроме того, если для дескриптора **multi_select** установлено значение "m", то лексемы в поле ввода, разделенные пробелами, рассматриваются как отдельные параметры.

Выполнение задач, связанных с окном диалога

Перед выполнением команды, заданной в дескрипторе **sm_cmd_hdr.cmd_to_exec**, SMIT создает дочерний процесс. Стандартный вывод сообщений об ошибках и стандартный вывод дочернего процесса обрабатываются в соответствии с инструкциями, содержащимися в дескрипторе **sm_cmd_hdr.exec_mode**. Затем в этом дочернем процессе SMIT выполняет функцию **setenv("ENV=")**, чтобы исключить возможность автоматического запуска команд из пользовательского файла **$HOME/.env** при запуске новой оболочки. Затем SMIT вызывает функцию **execl**, которая запускает оболочку **ksh**, используя в качестве значения параметра строку **ksh -c**.

SMIT формирует полные имена файлов протокола и флаги подробного режима, трассировки и отладки в той среде, в которой запускаются команды. Указанные значения задаются с помощью следующих переменных среды:

- **_SMIT_LOG_FILE**
- **_SMIT_SCRIPT_FILE**
- **_SMIT_VERBOSE_FLAG**
- **_SMIT_TRACE_FLAG**
- **_SMIT_DEBUG_FLAG**

Наличие или отсутствие соответствующего флага обозначается значением 1 или 0.

Кроме того, переменная среды **SMIT** содержит информацию об активной среде SMIT. Переменная среды **SMIT** может принимать следующие значения:

Значение	Среда SMIT
a	ASCII (текстовый) интерфейс SMIT
d	Распределенный интерфейс SMIT (DSMIT)
m	Оконный интерфейс SMIT (Motif)

Для того чтобы просмотреть текущие значения, вызовите после запуска SMIT функцию оболочки, а затем введите **env | grep SMIT**.

Вы можете отключить функциональную клавишу F9=Оболочка, установив переменную среды **SMIT_SHELL=n**.

Операции записи в файлы протокола должны выполняться в режиме добавления и сопровождаться немедленной записью содержимого буферных файлов на диск (если это не происходит автоматически).

Вы можете переопределить применяемый по умолчанию поток вывода дочернего процесса, указав в поле `sm_cmd_hdr.exec_mode` значение `"i"`. При такой настройке управление выводом передается задаче, поскольку процесс задачи просто наследует дескрипторы файла стандартного вывода сообщений об ошибках и файла стандартного вывода.

Если в поле `sm_cmd_hdr.exec_mode` будет указано значение `"e"`, то SMIT завершит работу, и вместо него будет запущена целевая задача.

Добавление задач в базу данных SMIT

При создании новых объектов в базу данных Инструмента управления системой (SMIT) рекомендуется сначала воспользоваться отдельной тестовой базой данных.

Процедура

Для создания тестовой базы данных выполните следующие действия:

1. Создайте каталог, который будет применяться при тестировании. Например, каталог `/home/smit/test` создается следующей командой:
   ```
   mkdir /home/smit /home/smit/test
   ```
2. Сделайте тестовый каталог текущим:
   ```
   cd /home/smit/test
   ```
3. Выберите тестовый каталог в качестве хранилища объектов по умолчанию. Для этого укажите в переменной среды **ODMDIR** значение **.** (текущий каталог):
   ```
   export ODMDIR= .
   ```
4. Создайте в тестовом каталоге новую базу данных SMIT:
   ```
   cp  /usr/lib/objrepos/sm_* .
   ```

Для добавления задач в базу данных SMIT:

1. Разработайте окно диалога для команды, которую должен будет сформировать SMIT.
2. Создайте иерархию меню и, при необходимости, иерархию промежуточных списков вариантов. Затем определите, каким образом эта иерархия должна быть включена в существующую базу данных SMIT. Если вы впервые создаете расширения базы данных SMIT, то для экономии времени и усилий рекомендуем вам придерживаться следующей процедуры:
   a. Запустите SMIT (вызвав команду **smit**), найдите меню, списки вариантов и окна диалогов, предназначенные для аналогичных задач, и выберите меню, в которые нужно добавить новую задачу.

b. Закройте SMIT, после чего удалите существующий файл протокола SMIT. Можно не удалять файл протокола, а ввести при следующем запуске SMIT команду **smit** с флагом **-l**. Это позволит вам выделить данные трассировки следующего сеанса работы SMIT.

c. Запустите SMIT с флагом **-t** и перейдите к меню, в которое вы собираетесь добавить новую задачу. При этом в протокол будут занесены идентификаторы всех объектов, к которым вы обращались.

d. Просмотрите файл протокола SMIT и определите идентификаторы классов объектов, используемых в качестве компонентов меню.

e. Найдите разделы файла настройки по ИД классов объектов с помощью команды **odmget**. Эти разделы можно использовать как образец при создании своих меню.

f. Найдите в файле протокола SMIT команды, используемые при запуске задач с помощью меню, и проверьте, применяются ли при этом специальные средства (например, сценарии **sed** и **awk**, функции оболочки **ksh**, переменные среды и т.п.). При вводе текста команд помните, что они обрабатываются дважды: первый раз - командой **odmadd**, а второй раз - оболочкой **ksh**. Будьте особенно осторожны с escape-символами, такими как \, или кавычками (' и "). Следует отметить, что параметры вывода команды **odmget** не всегда совпадают с параметрами ввода для команды **odmadd**, особенно при использовании escape-символов или значений параметров, занимающих несколько строк.

3. Добавьте программный код для объектов окна диалога, меню и списка в текстовый файл настройки объектов, который применяется командой **odmadd**.
4. С помощью команды **odmadd** добавьте объекты окна диалога, меню и списка вариантов в тестовую базу данных SMIT, заменив `test_stanzas` на имя текстового файла настройки объектов:

   ```
   odmadd test_stanzas
   ```

5. Проверьте и отладьте добавленные объекты, запустив SMIT с локальной тестовой базой данных:

   ```
   smit -o
   ```

Завершив тестирование, укажите в качестве хранилища объектов по умолчанию каталог **/etc/objrepos**, указав его в переменной среды **ODMDIR**:

```
export ODMDIR=/etc/objrepos
```

Отладка расширений базы данных SMIT

В этом разделе рассмотрен процесс отладки расширений базы данных SMIT.

Предварительные требования

1. Добавьте задачу в базу данных SMIT.
2. Протестируйте задачу.

Процедура

1. В зависимости от источника ошибки, укажите один из следующих флагов:
   - Если ошибка возникла в перечисленных ниже дескрипторах SMIT, выполните команду **smit -v**:
     - **cmd_to_list**
     - **cmd_to_classify**
     - **cmd_to_discover**
   - Если ошибки обнаружены в отдельных записях базы данных SMIT, выполните команду **smit-t**.
   - Для создания альтернативного файла протокола выполните команду **smit-l**. Этот файл применяется для сбора информации о текущем сеансе работы SMIT.
2. Измените базу данных SMIT, в которой хранится неправильная информация.
3. Протестируйте задачи SMIT еще раз.

Создание справки по новой задаче SMIT

Справка Инструмента управления системой (SMIT) представляет собой расширение программы SMIT.

С помощью справочной системы пользователь может получать информацию о компонентах SMIT, используемых для создания окон диалогов и меню. Разделы справки SMIT хранятся в базе данных аналогично исполняемым кодам программ SMIT. В SMIT предусмотрено два способа получения справочной информации:

Каждому из этих способов соответствует своя процедура получения информации из справочной базы данных SMIT.

Разделы справки man

Предварительные требования

Необходимо заранее создать новую задачу SMIT, для которой требуется справочная информация.

Процедура

1. С помощью любого редактора создайте файл, содержащий справочную информацию. Формат файла должен соответствовать команде **man**. В файле должен находиться один раздел справки.
2. Укажите заголовок файла с текстом справки в соответствии с правилами команды **man**.
3. Поместите файл с текстом справки в подкаталог **manual**.
4. Убедитесь, что созданный файл работает с командой **man**.
5. Найдите файл ASCII, в котором хранится объект настройки новой задачи SMIT.
6. В этом файле найдите поля дескрипторов справки.
7. В качестве значения поля `help_msg_loc` укажите заголовок файла с текстом справки. Этот заголовок также передается в качестве параметра команды **man**. Например:

   ```
   help_msg_loc = "xx", где "xx" = строка заголовка
   ```

   В этом примере команда **man** выполняется со строкой заголовка `xx`.
8. Остальные поля дескриптора справки оставьте пустыми.

Каталог сообщений

Предварительные требования

Необходимо заранее создать новую задачу SMIT, для которой требуется справочная информация.

Процедура

1. С помощью любого редактора создайте файл, содержащий сообщения справки. Формат файла **.msg** должен соответствовать правилам, которые применяются средством работы с сообщениями.

   Примечание: Можно использовать существующий файл **.msg**.
2. Для каждого сообщения задайте номер группы (Set #) и номер сообщения (MSG#). Они будут использоваться системой для поиска нужного пункта справки.
3. С помощью команды **gencat** преобразуйте файл **.msg** в файл **.cat**. Поместите файл **.cat** в каталог, который указан в переменной среды **NLSPATH**.
4. Протестируйте сообщения справки с помощью команды **dspmsg**.
5. Найдите файл ASCII, в котором хранится объект настройки новой задачи SMIT.
6. В этом файле найдите поля дескрипторов справки.
7. В каждом разделе найдите поле дескриптора справки `help_msg_id`. Укажите значения Set# и Msg# для сообщения из файла **.msg**. Формат этих значений должен соответствовать формату, определенному в Средствах работы с сообщениями. Например, для того чтобы получить сообщение #14 из группы #2, задайте:

   ```
   help_msg_id - "2,14"
   ```

8. Укажите в поле `help_msg_loc` дескриптора справки имя файла с текстом справки.
9. Остальные поля дескриптора справки оставьте пустыми.

Класс объектов sm_menu_opt (меню SMIT)

Объекты **sm_menu_opt** определяют отдельные пункты меню.

Меню представляет собой набор объектов с одинаковыми значениями дескрипторов **id** и объект заголовка **sm_menu_opt**, для которого значение **next_id** совпадает с **id** других объектов.

> **Примечание:** При программировании объекта данного объектного класса обозначайте пустые строки парой двойных кавычек (`""`), а пустые целые поля - нулем (0).

Существуют следующие дескрипторы объектов **sm_menu_opt**:

Дескриптор	Определение
id	Идентификатор или имя объекта. Значение поля **id** - строка длиной до 64 символов. Идентификаторы должны быть уникальны как в приложении, так и в используемой базе данных SMIT. (Связанная информация приведена в определениях **next_id** и **alias** для данного объекта).
id_seq_num	Расположение данного пункта по отношению к остальным пунктам меню. В этом поле хранятся объекты **sm_menu_opt** без заголовков. Значение поля **id_seq_num** представляет собой строку длиной до 16 символов.
next_id	Команда быстрого доступа к следующему меню, если для дескриптора **next_type** данного объекта задано значение `"m"` (меню). Идентификатор меню **next_id** должен быть уникален как в приложении, так и в используемой базе данных SMIT. Все объекты **sm_menu_opt** (не псевдонимы), для которых значения **id** совпадают со значением **next_id**, образуют набор списков вариантов для данного меню. Значение **next_id** - строка длиной до 64 символов.
текст	Текст описания задачи, связанной с пунктом меню. Значение поля **text** - строка длиной до 1024 символов. Строка может быть отформатирована и может содержать символы новой строки (`\n`).
text_msg_file	Неполное имя файла для каталога Средств работы с сообщениями, применяемого для строки **text**. Значение поля **text_msg_file** - строка длиной до 1024 символов. Средства работы с сообщениями можно применять для создания каталогов сообщений прикладных программ. Если вы не будете использовать Средства работы с сообщениями, укажите в этом поле `""`.
text_msg_set	Идентификатор набора Средств работы с сообщениями для строки **text**. Эти идентификаторы могут применяться для указания подкаталогов одного и того же каталога. Поле **text_msg_set** - целого типа. Если вы не будете использовать Средства работы с сообщениями, укажите в этом поле нулевое значение.
text_msg_id	Идентификатор Средства работы с сообщениями для строки **text**. Поле **text_msg_id** - целого типа. Если вы не будете использовать Средства работы с сообщениями, укажите в этом поле нулевое значение.
next_type	Тип объекта, который будет показан при выборе данного пункта. Для этого флага допустимы следующие значения: **"m"** Меню; будет показано меню (**sm_menu_opt**). **"d"** Окно диалога; будет показано окно диалога (**sm_cmd_hdr**). **"n"** Имя; будет показан список вариантов (**sm_name_hdr**). **"i"** Информация; данный объект применяется для вставки в меню пустых или разделительных строк, а также пунктов меню, предназначенных исключительно для получения справочной информации, ссылка на которую содержится в дескрипторе **help_msg_loc** данного объекта.
alias	Указывает, является ли значение дескриптора **id** данного объекта меню псевдонимом другой команды быстрого доступа, определенной в поле `next_id` данного объекта. Для дескриптора **alias** объекта меню необходимо задать значение `"n"`.
help_msg_id	Поле, в котором задается номер набора сообщений Средства работы с сообщениями и (через запятую) идентификатор сообщения, либо строка цифр, совпадающая с тегом идентификатора SMIT.
help_msg_loc	Имя файла, передаваемого в качестве параметра команде **man** для поиска текста справки, либо имя файла, содержащего текст справки. Значение **help_msg_loc** - строка длиной до 1024 символов.
help_msg_base	Полное имя библиотеки, в которой SMIT ищет имена файлов, связанные с нужным томом справки.
help_msg_book	Содержит строку с именем файла из библиотеки, указанной в **help_msg_base**.

Класс объектов sm_menu_opt, предназначенный для создания псевдонимов

Объект **sm_menu_opt** определяет псевдоним SMIT.

Ниже перечислены дескрипторы классов объектов **sm_menu_opt** и их возможные значения:

Дескриптор	Определение
id	Идентификатор или имя команды быстрого доступа или ее псевдонима. Значение поля **id** - строка длиной до 64 символов. Идентификаторы должны быть уникальны как в приложении, так и в базе данных SMIT, в которой они используются.
id_seq_num	Задайте в этом поле пустую строку ("").
next_id	Задает **id_seq_num** объекта меню, на который указывает данный псевдоним. Значение **next_id** - строка длиной до 64 символов.
текст	Задайте в этом поле пустую строку ("").
text_msg_file	Задайте в этом поле пустую строку ("").
text_msg_set	Задайте в этом поле нулевое значение (0).
text_msg_id	Задайте в этом поле нулевое значение (0).
next_type	Тип окна для команды быстрого доступа. Значение **next_type** - строчная константа. Для этого флага допустимы следующие значения: **"m"** Меню; в поле next_id указана команда быстрого доступа к меню. **"d"** Окно диалога; в поле next_id указана команда быстрого доступа к окну диалога. **"n"** Имя; в поле next_id указана команда быстрого доступа к окну списка вариантов.
alias	Определяет данный объект как псевдоним для команды быстрого доступа. Для этого в дескрипторе **alias** должно быть задано значение "y" (да).
help_msg_id	Задайте в этом поле пустую строку ("").
help_msg_loc	Задайте в этом поле пустую строку ("").
help_msg_base	Задайте в этом поле пустую строку ("").
help_msg_book	Задайте в этом поле пустую строку ("").

Класс объектов sm_name_hdr (заголовок списков вариантов SMIT)

Для определения окна со списком вариантов применяются два объекта: **sm_name_hdr**, задающий заголовок окна и другую информацию, и **sm_cmd_opt**, определяющий тип получаемых данных.

> **Примечание:** При программировании объекта данного объектного класса обозначайте пустые строки парой двойных кавычек (""), а пустые целые поля - нулем (0).

Если в меню SMIT Заголовок списка (**sm_name_hdr**) с типом "c" при вводе значений используется двоеточие (:), например, tty:0, то программа SMIT вставит перед двоеточием #! (знак фунта и восклицательный знак), чтобы не путать двоеточие с разделителем полей. После того как SMIT проанализирует оставшуюся часть значения, символы #! будут удалены, а значение записано в дескриптор **cmd_to_classify**. При последующем добавлении данных в дескриптор **cmd_to_classify** необходимо снова вставить #! перед :

Существуют следующие дескрипторы класса объектов **sm_name_hdr**:

Дескриптор	Определение
id	Идентификатор или имя объекта. В поле id можно указать идентификатор команды быстрого доступа, если только для **has_name_select** не установлено значение "y" (да). Значение поля **id** - строка длиной до 64 символов. Идентификатор должен быть уникален как в приложении, так и в системе.
next_id	Задает объект заголовка следующего окна; в этом поле указано значение поля id объекта **sm_cmd_hdr** или объекта **sm_name_hdr**, следующего за данным списком вариантов. Описанное ниже поле next_type определяет класс объекта. Значение **next_id** - строка длиной до 64 символов.
option_id	Задает содержимое списка вариантов; соответствует значению поля id объекта **sm_cmd_opt**. Значение поля **option_id** - строка длиной не более 64 символов.

Дескриптор	Определение
has_name_select	Указывает, будет ли перед данным окном выдаваться окно со списком вариантов. Для этого флага допустимы следующие значения: **"" или "n"** Нет; это значение по умолчанию. **Идентификатор** данного объекта может использоваться в качестве команды быстрого доступа, даже если перед ним выдается окно со списком вариантов. **"y"** Да; данному объекту может предшествовать список выбора. Если указано это значение, то **id** данного объекта не может применяться в качестве команды быстрого доступа к соответствующему окну диалога.
name	Текст для заголовка окна со списком вариантов. Значение поля **name** - строка длиной не более 1024 символов. Строка может быть отформатирована и может содержать символы новой строки (\n).
name_msg_file	Неполное имя файла для каталога Средств работы с сообщениями, применяемого для строки **name**. Значение поля **name_msg_file** - строка длиной не более 1024 символов. Средства работы с сообщениями можно применять для создания каталогов сообщений прикладных программ.
name_msg_set	Идентификатор набора Средств работы с сообщениями для строки **name**. Эти идентификаторы могут применяться для указания подкаталогов одного и того же каталога. Поле **name_msg_set** - целого типа.
name_msg_id	Идентификатор Средства работы с сообщениями для строки **name**. Поле **name_msg_id** - целого типа.
type	Способ обработки списка вариантов. Значение поля **type** - строка, состоящая из одного символа. Для этого флага допустимы следующие значения: **"" или "j"** Следующий идентификатор; за данным объектов всегда следует объект, определяемый значением дескриптора **next_id**. Дескриптор **next_id** - это строка максимальной длины, инициализированная в программе при ее создании. **"r"** Конкатенация исходного (необработанного) имени; в этом случае дескриптор **next_id** частично определяется при написании программы, а частично - при ее выполнении, когда пользователь вводит недостающую информацию. Выполняется конкатенация значения дескриптора **next_id**, определенного в программе, и значения, указанного пользователем. Полученное в результате значение дескриптора **id** применяется для поиска следующего объекта (окна диалога или списка вариантов). **"c"** Конкатенация обработанного имени; для получения дополнительной информации необходима обработка значения, выбранного пользователем. Это значение передается в команду, указанную в дескрипторе **cmd_to_classify**, после чего вывод команды объединяется со значением дескриптора **next_id** для создания дескриптора **id**, который используется для поиска следующего объекта (окна диалога или списка вариантов).
ghost	Указывает, следует ли показывать окно со списком вариантов или только всплывающую панель, которая была создана командой, указанной в поле cmd_to_list. Значение **ghost** - строчная константа. Для этого флага допустимы следующие значения: **"" или "n"** Нет; показывать окно со списком вариантов. **"y"** Да; показывать только всплывающую панель, которая была создана командой, указанной в полях cmd_to_list и cmd_to_list_postfix связанного объекта **sm_cmd_opt**. Если значение в поле **cmd_to_list** не указано, то SMIT считает данный объект виртуальным, выполняет команду **cmd_to_classify** и продолжает обработку.

Дескриптор	Определение
cmd_to_classify	Задает текст команды, которая при необходимости будет использоваться для классификации значения поля `name` объекта **sm_cmd_opt**. Значение поля **cmd_to_classify** представляет собой строку длиной до 1024 символов. Входные данные для **cmd_to_classify** (исходное имя) считываются из поля `entry`, а на выходе команда **cmd_to_classify** выдает обработанное имя. В версиях AIX, предшествующих 4.2.1, с помощью **cmd_to_classify** можно было создавать только одно значение. Если это значение содержит двоеточие, оно автоматически пропускается. В AIX 4.2.1 и выше с помощью **cmd_to_classify** можно создать сразу несколько значений, причем данная команда использует двоеточие как ограничитель. Если вам нужны значения с символами двоеточий, то задавайте их вручную.
cmd_to_classify_postfix	Постфикс, который анализируется и добавляется к тексту команды, заданному в поле `cmd_to_classify`. Значение поля **cmd_to_classify_postfix** представляет собой строку длиной до 1024 символов.
raw_field_name	Альтернативное имя для исходного значения. Значение поля **raw_field_name** представляет собой строку длиной до 1024 символов. Значение по умолчанию - `"_rawname"`.
cooked_field_name	Альтернативное имя для обработанного значения. Значение поля **cooked_field_name** представляет собой строку длиной до 1024 символов. Значение по умолчанию - `"cookedname"`.
next_type	Тип окна, показываемого после списка вариантов. Для этого флага допустимы следующие значения: **"n"** Имя; будет показано окно со списком вариантов. Подробная информация приведена в описании поля **next_id** (см. выше). **"d"** Окно диалога; будет показано окно диалога. Подробная информация приведена в описании поля **next_id** (см. выше).
help_msg_id	Поле, в котором задается номер набора сообщений Средства работы с сообщениями и (через запятую) идентификатор сообщения, либо строка цифр, совпадающая с тегом идентификатора SMIT.
help_msg_loc	Имя файла, передаваемого в качестве параметра команде **man** для поиска текста справки, либо имя файла, содержащего текст справки. Значение **help_msg_loc** - строка длиной до 1024 символов.
help_msg_base	Полное имя библиотеки, в которой SMIT ищет имена файлов, связанные с нужным томом справки.
help_msg_book	Содержит строку с именем файла из библиотеки, указанной в **help_msg_base**.

Класс объектов sm_cmd_opt (опции команд окна диалога/списка вариантов SMIT)

Каждый объект в окне диалога, за исключением объекта заголовка, соответствует флагу, опции или атрибуту команды, выполняемой в этом окне.

Для каждого окна диалога SMIT создается один или несколько объектов; с динамическим окном диалога может не быть связано ни одного объекта опций команды. Окно списка вариантов состоит из одного объекта, определяющего заголовок списка, и из одного объекта, определяющего опции команды.

> **Примечание:** При программировании объекта данного объектного класса обозначайте пустые строки парой двойных кавычек (`""`), а пустые целые поля - нулем (`0`).

Для определения опций команды окна диалога и команды списка вариантов применяется объект **sm_cmd_opt**. Ниже перечислены дескрипторы класса объектов **sm_cmd_opt** и их функции:

Дескриптор	Функция
id	Идентификатор или имя объекта. **Идентификатор** объекта связанного окна диалога или заголовка списка вариантов может использоваться как команда быстрого доступа к этому или к другим объектам окон диалога в данном окне. Значение поля **id** - строка длиной до 64 символов. Для всех объектов одного и того же окна диалога необходимо указать одинаковый идентификатор. Идентификатор должен быть уникален как в приложении, так и в базе данных SMIT, в которой он используется.
id_seq_num	Расположение данного элемента по отношению к остальным элементам в окне диалога; строка, задаваемая в данном поле, определяет, в каком порядке объекты **sm_cmd_opt** располагаются по отношению к другим элементам окна диалога. Значение поля **id_seq_num** представляет собой строку длиной до 16 символов. Если объект - это компонент окна диалога, то в данном поле нельзя указывать строку `"0"`. Если объект - это элемент окна списка вариантов, то в дескрипторе **id_seq_num** должно быть задано нулевое значение.
disc_field_name	Строка, которая должна совпадать с одним из полей имен в выводе команды **cmd_to_discover** в заголовке связанного окна диалога. Значение поля **disc_field_name** - строка длиной до 64 символов. Значение дескриптора **disc_field_name** можно определять не только с помощью команды **cmd_to_discover** в связанном объекте заголовка, но и с помощью исходного или измененного имени, которое будет выбираться из предложенного списка вариантов. Если имя будет выбираться путем выбора из списка вариантов, то в дескрипторе нужно указать значение `"_rawname"`, `"_cookedname"`, или же соответствующее имя `sm_name_hdr.cooked_field_name` либо `sm_name_hdr.raw_field_name`, если оно переопределяет имя по умолчанию.
name	Строка, которая будет показана в окне диалога или в списке вариантов как имя поля. Это часть объекта, содержащая вопрос или приглашение, который будет показан на экране. Эта строка описывает на естественном языке флаг, опцию или параметр команды, указанной в поле `cmd_to_exec` связанного объекта заголовка окна диалога. Значение поля **name** - строка длиной не более 1024 символов.
name_msg_file	Неполное имя файла для каталога Средств работы с сообщениями, применяемого для строки **name**. Значение поля **name_msg_file** - строка длиной не более 1024 символов. Средства работы с сообщениями можно применять для создания каталогов сообщений прикладных программ. Если вы не планируете использовать Средства работы с сообщениями, укажите в этом поле пустую строку (`""`).
name_msg_set	Идентификатор набора Средств работы с сообщениями для строки **name**. Поле **name_msg_set** - целого типа. Если Средства работы с сообщениями применяться не будут, задайте в этом поле нулевое значение (`0`).
name_msg_id	Идентификатор сообщения из Средств работы с сообщениями для строки **name**. Поле **name_msg_id** - целого типа. Если Средства работы с сообщениями применяться не будут, задайте в этом поле нулевое значение (`0`).
op_type	Тип вспомогательной операции, поддерживаемой для данного поля. Значение **op_type** - строчная константа. Для этого флага допустимы следующие значения: **""** или **"n"** - это значение по умолчанию. Для данного поля не предусмотрено вспомогательных операций (выбора из списка или из кольцевого списка). **"l"** - Поддерживается операция выбора из списка. При нажатии клавиши **F4=Список** на экране появляется всплывающее меню со списком элементов, созданным командой, указанной в поле `cmd_to_list` данного объекта. **"r"** - Поддерживается операция выбора из кольцевого списка. Строка, заданная в поле `disp_values` или `aix_values`, интерпретируется как разделенный запятыми набор допустимых записей. Пользователь может передвигаться по списку, нажимая клавишу Tab или Backtab. В этом случае клавиша меню SMIT **F4=Список** остается допустимой, поскольку при необходимости SMIT преобразует кольцевой список в обычный. При задании значения в поле **op_type** можно использовать как прописные символы (`"N"`, `"L"` и `"R"`), так и строчные символы (`"n"`, `"l"` и `"r"`). Если значения указаны в верхнем регистре, а при выполнении команды **cmd_to_exec** возвращается значение `0`, то соответствующие поля ввода будут очищены, а их значения заменены пустыми строками.

Дескриптор	Функция
entry_type	Тип значения, задаваемого в поле ввода. Значение в поле **entry_type** - строка. Для этого флага допустимы следующие значения: **""** или **"n"** - ввод запрещен; текущее значение нельзя изменить с помощью клавиатуры. Это поле служит только для просмотра информации. **"t"** - текстовое поле; можно вводить буквенно-цифровые символы. **"#"** - числовое поле; значение в данном поле должно состоять только из цифр (0, 1, 2, 3, 4, 5, 6, 7, 8 или 9). Первым символом может быть знак **-** (минус) или + (плюс). **"x"** - шестнадцатеричное поле; можно вводить только шестнадцатеричные числа. **"f"** - имя файла; необходимо ввести имя файла. **"r"** - исходный текст; можно вводить буквенно-цифровые символы. Начальные и конечные пробелы учитываются и не удаляются при считывании значения поля.
entry_size	Ограничение на число символов, указываемых в поле ввода. Поле **entry_size** - целого типа. Значение по умолчанию - 0 - соответствует максимальной допустимой длине для указываемого типа.
required	Указывает, должно ли значение поля передаваться команде **cmd_to_exec**, указанной в объекте заголовка окна диалога. Значение поля **required** - строка. Если объект - элемент окна списка вариантов, то в поле required обычно указывается пустая строка (""). Если объект - элемент окна диалога, то допустимы следующие значения: **""** или **"n"** - нет; опция добавляется к тексту команды в дескрипторе **cmd_to_exec** только в том случае, если показанное на экране начальное значение было изменено пользователем. Это значение по умолчанию. **"y"** - да; значения поля prefix и поля ввода всегда передаются в команду **cmd_to_exec**. **"+"** - значения поля prefix и поля ввода всегда передаются в команду **cmd_to_exec**. Поле ввода должно содержать хотя бы один отличный от пробела символ, в противном случае SMIT не разрешит пользователю запустить задачу. **"?"** - значения поля prefix и поля ввода передаются в команду cmd_to_exec только в том случае, если в поле ввода задано непустое значение.
prefix	В простейшем случае определяет флаг, передаваемый вместе со значением поля ввода команде **cmd_to_exec**, которая была указана в объекте заголовка окна диалога. Значение поля **prefix** - строка длиной до 1024 символов. Смысл этого поля зависит от того, какие значения указаны в полях required, prefix и в связанном поле entry. **Примечание:** Если в поле prefix задано значение -, то содержимое соответствующего поля ввода добавляется в конец команды **cmd_to_exec**. Если в поле prefix указано значение -' (двойной дефис и одинарная кавычка), то в конец команды **cmd_to_exec** добавляется содержимое соответствующего поля ввода, заключенное в одинарные кавычки.
cmd_to_list_mode	Определяет, какая часть элемента списка должна использоваться. Список создается командой, которая указана в поле cmd_to_list данного объекта. Значение **cmd_to_list_mode** - строка, состоящая из одного символа. Для этого флага допустимы следующие значения: **""** или **"a"** - загружать все поля. Это значение по умолчанию. **"1"** - загружать первое поле. **"2"** - загружать второе поле. **"r"** - диапазон; команда, указанная в поле cmd_to_list, вместо списка создает диапазон (например, **1..99**). Диапазоны предназначены только для просмотра информационных данных; они показаны в неизменяемом всплывающем списке.
cmd_to_list	Текст команды, которая используется для получения списка допустимых значений поля ввода. Значение поля **cmd_to_list** - строка длиной до 1024 символов. Значения в списке, выдаваемом этой командой, должны отделяться друг от друга символом новой строки (\n).

Дескриптор	Функция
cmd_to_list_postfix	Постфикс, который анализируется и добавляется к тексту команды, заданной в поле `cmd_to_list` объекта окна диалога. Значение поля **cmd_to_list_postfix** - строка длиной до 1024 символов. Если первая строка начинается с пробела и следующего за ним символа # (знака фунта), то эту запись нельзя будет выбрать из списка. Такие строки применяются для заголовков столбцов. Последующие строки, начинающиеся со знака #, перед которым может стоять пробел, рассматриваются как комментарии и строки продолжения.
multi_select	Указывает, может ли пользователь выбирать сразу несколько вариантов из списка возможных значений, создаваемого командой, которая указана в поле `cmd_to_list` объекта окна диалога. Значение **multi_select** - строка. Для этого флага допустимы следующие значения: **""** - нет; пользователь может выбрать из списка только одно значение. Это значение по умолчанию. **","** - да; пользователь может выбрать несколько элементов из списка. При формировании команды элементы разделяются запятыми. **"y"** - да; пользователь может выбрать несколько элементов из списка. При формировании команды перед списком выбранных элементов вставляется префикс опции. **"m"** - да; пользователь может выбрать несколько элементов из списка. При формировании команды перед каждым выбранным элементом вставляется префикс опции.
value_index	Отсчитываемый от нуля индекс массива полей `disp_value` для кольцевого списка опций. Номер **value_index** указывает на значение, которое будет показано в пользовательском поле ввода как значение по умолчанию. Поле **entry_size** - целого типа.
disp_values	Массив значений в кольцевом списке опций, показываемом пользователю. Значение `disp_values` - строка длиной до 1024 символов. Значения полей указываются через запятую (,), без пробелов.
values_msg_file	Имя файла каталога Средства работы с сообщениями, применяемого для значений `disp_values`, если они задаются на стадии разработки. Значение поля `values_msg_file` - строка длиной не более 1024 символов. Средства работы с сообщениями можно применять для создания каталогов сообщений прикладных программ.
values_msg_set	Идентификатор набора Средств работы с сообщениями для значений полей `disp_values`. Если Средства работы с сообщениями применяться не будут, укажите нулевое значение.
values_msg_id	Идентификатор сообщения из Средств работы с сообщениями для значений полей `disp_values`. Если Средства работы с сообщениями применяться не будут, укажите нулевое значение.
aix_values	Этот дескриптор используется в том случае, если каждый элемент массива значений кольцевого списка опций соответствует элементу массива **disp_values** в той же позиции, а значения, заданные на естественном языке в **disp_values**, отличаются от фактических опций, используемых для данной команды. Значение поля `aix_values` - строка длиной не более 1024 символов.
help_msg_id	Поле, в котором задается номер набора сообщений Средства работы с сообщениями и (через запятую) идентификатор сообщения, либо строка цифр, совпадающая с тегом идентификатора SMIT.
help_msg_loc	Имя файла, передаваемого в качестве параметра команде **man** для поиска текста справки, либо имя файла, содержащего текст справки. Значение **help_msg_loc** - строка длиной до 1024 символов.
help_msg_base	Полное имя библиотеки, в которой SMIT ищет имена файлов, связанные с нужным томом справки.
help_msg_book	Содержит строку с именем файла из библиотеки, указанной в **help_msg_base**.

Класс объектов sm_cmd_hdr (заголовок окна диалога SMIT)

Заголовок окна диалога хранится в объекте **sm_cmd_hdr**. Этот объект создается для каждого окна диалога и указывает на объекты опций команд окна диалога, связанные с данным окном.

Примечание: При создании объекта данного класса обозначайте пустые строки парой двойных кавычек (`""`) и присваивайте неиспользуемым целочисленным полям нулевое значение.

Существуют следующие дескрипторы класса **sm_cmd_hdr**:

Дескриптор	Функция
id	Идентификатор или имя объекта. Значение поля **id** - строка длиной до 64 символов. Если с данным окном диалога не связан список вариантов, то поле `id` можно применять в качестве идентификатора команды быстрого доступа. Идентификатор должен быть уникален как в приложении, так и в системе.
option_id	**Идентификатор** объектов **sm_cmd_opt** (полей окна диалога), на которое указывает данный заголовок. Значение поля **option_id** - строка длиной не более 64 символов.
has_name_select	Указывает, будет ли перед данным окном выдаваться меню или окно со списком вариантов. Для этого флага допустимы следующие значения: **"" или "n"** Нет; это значение по умолчанию. **"y"** Да; перед этим объектом будет выдаваться список вариантов. Если указано это значение, то **id** данного объекта не может применяться в качестве команды быстрого доступа к соответствующему окну диалога.
name	Текст, который будет выдаваться в качестве заголовка окна диалога. Значение поля **name** - строка длиной не более 1024 символов. Текст представляет собой описание задачи, выполняемой данным окном диалога. Строка может быть отформатирована и может содержать символы новой строки (**\n**).
name_msg_file	Неполное имя файла для каталога Средств работы с сообщениями, применяемого для строки **name**. Значение поля **name_msg_file** - строка длиной не более 1024 символов. Средства работы с сообщениями можно применять для создания каталогов сообщений прикладных программ.
name_msg_set	Идентификатор набора Средств работы с сообщениями для строки **name**. Эти идентификаторы могут применяться для указания подкаталогов одного и того же каталога. Поле **name_msg_set** - целого типа.
name_msg_id	Идентификатор Средства работы с сообщениями для строки **name**. Идентификаторы сообщений можно создавать средствами выбора сообщений, входящими в состав Средств работы с сообщениями. Поле **name_msg_id** - целого типа.
cmd_to_exec	Начальная часть командной строки, которая состоит из самой команды, выполняющей требуемое действие, или из команды и постоянного набора некоторых опций. Остальные опции добавляются автоматически при работе пользователя с объектами опций команды (**sm_cmd_opt**), связанными с данным окном диалога. Значение поля **cmd_to_exec** - строка длиной до 1024 символов.
ask	Указывает, должна ли система предлагать пользователю еще раз проверить выбранные варианты, перед тем как приступить к выполнению задачи. Для этого флага допустимы следующие значения: **"" или "n"** Нет; не требовать от пользователя подтверждения выбора; выполнение задачи начинается при фиксации изменений в окне диалога. Для дескриптора **ask** это значение по умолчанию. **"y"** Да; выполнение задачи начинается только после того, как она будет подтверждена пользователем. Запрос на подтверждение особенно полезен в тех случаях, когда выполняются задачи по удалению каких-либо объектов, причем удаляемые ресурсы трудно или невозможно восстановить, или когда с задачей не связано окно диалога (т.е. в поле `ghost` указано значение `"y"`).

Дескриптор	Функция
exec_mode	Задает режим обработки стандартного ввода, стандартного вывода и файла **stderr** при выполнении задачи. Значение **exec_mode** - строкового типа. Для этого флага допустимы следующие значения: **"" или "p"** Конвейерный режим; для дескриптора **exec_mode** это значение по умолчанию. Стандартный вывод команды и файл **stderr** перенаправляются в программу SMIT. Стандартным выводом команды управляет SMIT. После завершения задачи выходные данные сохраняются и могут быть просмотрены пользователем в режиме прокрутки. При необходимости вывод команды можно прокручивать на экране и во время выполнения задачи. **"n"** Конвейерный режим без прокрутки; аналогичен режиму `"p"` за исключением того, что нельзя прокручивать информацию на экране во время выполнения задачи. При запуске задачи появляется окно вывода, которое остается на экране все время, пока выполняется задача. После завершения задачи выходные данные сохраняются и могут быть просмотрены пользователем в режиме прокрутки. **"i"** Режим наследования; стандартный ввод, вывод и **stderr** команды наследуется дочерним процессом, в котором выполняется задача. В этом режиме управление вводом и выводом передается выполняемой команде. Рекомендуется указывать это значение для команд, которым необходимо записывать данные в файл **/dev/tty**, выполнять адресацию курсора или работать с библиотеками **libcur** или **libcurses**. **"e"** Режим "выход/выполнение"; в этом режиме SMIT запускает команду в текущем процессе (вызывает функцию **execl**), после чего программа SMIT завершается. Рекомендуется указывать этот режим для команд, несовместимых с программой SMIT (которые, например, изменяют режимы просмотра или размеры шрифтов). Будет выдано предупреждение о том, что перед запуском команды программа SMIT будет завершена. **"E"** То же, что и **"e"**, но без выдачи предупреждения о выходе из программы SMIT. **"" или "p"** Режимы с восстановлением предыдущего состояния. Аналогично режимам `"p"`, `"n"` и `"i"`, за исключением того, что если при выходе из команды **cmd_to_exec** возвращается значение `0`, то SMIT переходит в ближайшее предыдущее меню, к списку вариантов (если он существует) или в командную строку.
ghost	Указывает, должно ли быть скрыто окно диалога, обычно показываемое на экране. Значение **ghost** - строчная константа. Для этого флага допустимы следующие значения: **"" или "n"** Нет; окно диалога, связанное с задачей, будет показано на экране. Это значение по умолчанию. **"y"** Да; окно диалога, связанное с задачей, не показывается, так как от пользователя больше не требуется никакой информации. Команда, указанная в дескрипторе **cmd_to_exec**, начнет выполняться сразу, как только пользователь выберет данную задачу.
cmd_to_discover	Задает текст команды, используемой для получения значений по умолчанию или текущих значений для обрабатываемого объекта. Значение **cmd_to_discover** - строка длиной до 1024 символов. Команда выполняется перед открытием окна диалога; считываются ее выходные данные. Выходные данные должны быть в формате с двоеточиями.
cmd_to_discover_postfix	Постфикс, который анализируется и добавляется к тексту команды, заданному в поле `cmd_to_discover`. Значение **cmd_to_discover_postfix** - строка длиной не более 1024 символов.
help_msg_id	Поле, в котором задается номер набора сообщений Средства работы с сообщениями и (через запятую) идентификатор сообщения, либо строка цифр, совпадающая с тегом идентификатора SMIT.

Дескриптор	Функция
help_msg_loc	Имя файла, передаваемого в качестве параметра команде **man** для поиска текста справки, либо имя файла, содержащего текст справки. Значение **help_msg_loc** - строка длиной до 1024 символов.
help_msg_base	Полное имя библиотеки, в которой SMIT ищет имена файлов, связанные с нужным томом справки.
help_msg_book	Содержит строку с именем файла из библиотеки, указанной в **help_msg_base**.

Пример программы SMIT

Программу, приведенную в данном примере, можно применять для создания собственных файлов настройки.

Если добавить эти файлы в каталог SMIT, который поставляется вместе с операционной системой, то к ним можно будет обратиться с помощью пункта **Приложения** главного меню SMIT. Здесь представлены работающие версии разделов описания, за исключением Примера 3, в котором не устанавливается никаких языков.

```
#------------------------------------------------------------------
# Введение:
# Прежде чем создавать новую базу данных SMIT, необходимо решить,
# в каком месте в иерархии меню будет находиться меню вашего приложения.
# Новое меню будет ссылаться на другие меню, заголовки и
# окна диалогов. Мы, например, вставляем указатель на новое
# меню вслед за опцией меню "Приложения". Значение next_id для пункта
# меню "Приложения" равно "apps", поэтому мы начинаем с создания
# menu_opt со значением id, равным "apps".
#------------------------------------------------------------------
sm_menu_opt:
      id           = "apps"
      id_seq_num   = "010"
      next_id      = "demo"
      text         = "Примеры программ SMIT"
      next_type    = "m"

sm_menu_opt:
      id              = "demo"
      id_seq_num   = "010"
      next_id                  = "demo_queue"
      text         = "Пример 1: Добавить очередь печати"
   next_type                = "n"

sm_menu_opt:
      id             = "demo"
   id_seq_num           = "020"
      next_id         = "demo_mle_inst_lang_hdr"
      text            = "Пример 2: Добавить язык для уже установленного приложения"
   next_type                = "n"

#----
# Поскольку demo_mle_inst_lang_hdr - описательное имя, но его трудно
# запомнить, то для ссылки на это окно можно создать более простой
# псевдоним.
#----
sm_menu_opt:
      id             = "demo_lang"
      next_id         = "demo_mle_inst_lang_hdr"
   next_type                = "n"
      alias           = "y"

sm_menu_opt:
   id_seq_num           = "030"
      id             = "demo"
      next_id         = "demo_lspv"
      text            = "Пример 3: Оглавление физического тома"
```

```
      text_msg_file   = "smit.cat"
   next_type                = "n"

sm_menu_opt:
      id_seq_num      = "040"
      id              = "demo"
      next_id         = "demo_date"
      text            = "Пример 4: Изменить/показать дату, время"
      text_msg_file   = "smit.cat"
   next_type                = "n"

#-------------------------------------------------------------------
# Пример 1
# ------
# Цель: Добавить очередь печати. Если пакет printers.rte не установлен,
#       то он должен устанавливаться автоматически. Если пользователь работает
#       со MSMIT (SMIT с оконным интерфейсом), то для выполнения этой задачи
#       будет запускаться графическая программа. В противном случае
#       управление передается задаче очереди печати SMIT.
#
# Вопросы:     1. Обработанный вывод & cmd_to_classify
#              2. Переменная среды SMIT (msmit или ascii)
#              3. name_hdr для динамического окна
#              4. name_hdr для потомка динамического окна
#              5. Создание опции "OK / Отмена"
#              6. dspmsg для преобразований
#              7. Режим exit/exec
#              8. id_seq_num для опции name_hdr
#-------------------------------------------------------------------
#----
# Вопросы: 1,4
# Следует отметить, что значение next_id совпадает со значением id.
# Помните, что обработанный вывод cmd_to_classify добавляется в next_id,
# так как задан тип "c". Поэтому значением для next_id будет либо
# demo_queue1, либо demo_queue2. Вывод name_hdr не показывается,
# а значение cmd_to_list в demo_queue_dummy_opt не указано,
# т. е. окно с данным name_hdr будет динамическим.
#----
sm_name_hdr:
      id                        = "demo_queue"
      next_id                   = "demo_queue"
      option_id        = "demo_queue_dummy_opt"
      name                      = "Добавить очередь печати"
   name_msg_file             = "smit.cat"
      name_msg_set              = 52
      name_msg_id               = 41
      type                      = "c"
      ghost = "y"
      cmd_to_classify           = "\
x()
{
    # Проверяется, установлен ли файл принтера.
    lslpp -l printers.rte 2>/dev/null 1>/dev/null
    if [[ $? != 0 ]]
    то
      echo 2
    else
      echo 1
    fi
}
x"
   next_type                = "n"

#----
# Вопросы: 2,4
# Определив, что программное обеспечение принтера установлено, мы
# хотим узнать, запускать ли для данной задачи программу GUI,
```

```
# или переходить в текстовое окно SMIT. Для этого мы проверяем
# значение переменной среды SMIT, которое равно "m" для оконного
# (Motif) или "a" для текстового (ascii) режима. Здесь вывод
# cmd_to_classify снова передается в next_id.
#----
sm_name_hdr:
      id                        = "demo_queue1"
      next_id                   = "mkpq"
      option_id        = "demo_queue_dummy_opt"
   has_name_select        = ""
      ghost = "y"
   next_type                = "n"
      type                      = "c"
   cmd_to_classify          = "\
gui_check()
{
   if [ $SMIT = \"m\" ]; then
     echo gui
   fi
}
      gui_check"

sm_name_hdr:
      id               = "mkpqgui"
      next_id          = "invoke_gui"
   next_type = "d"
      option_id        = "demo_queue_dummy_opt"
      ghost = "y"

#----
# Вопрос: 7
# Примечание: значение exec_mode для команды равно "e"; это означает,
# что перед запуском cmd_to_exec программа SMIT завершается.
#----
sm_cmd_hdr:
      id               = "invoke_gui"
      cmd_to_exec      = "/usr/bin/X11/xprintm"
      exec_mode        = "e"
      ghost = "y"

sm_cmd_opt:
      id                            = "demo_queue_dummy_opt"
      id_seq_num                    = 0

#----
# Вопросы: 3,5
# Программное обеспечение принтера не установлено. Установите
# его и вернитесь к demo_queue1, чтобы проверить значение переменной
# среды SMIT. Это name_hdr для динамического окна. cmd_to_list для
# sm_cmd_opt будет немедленно появляться на экране в виде всплывающей опции,
# вместо того, чтобы ждать, пока пользователь введет данные.
# В этом динамическом окне cmd_opt задает окно "OK/Отмена",
# в котором пользователю требуется просто нажать клавишу Return.
#----
sm_name_hdr:
      id                        = "demo_queue2"
      next_id                   = "demo_queue1"
      option_id                 = "demo_queue_opt"
      name                      = "Добавить очередь печати"
   name_msg_file             = "smit.cat"
      name_msg_set              = 52
      name_msg_id               = 41
      ghost = "y"
   cmd_to_classify           = "\
install_printers ()
{
```

```
  # Установить пакет принтера.
  /usr/lib/assist/install_pkg \"printers.rte\" 2>&1 >/dev/null
  if [[ $? != 0 ]]
  то
    echo "Ошибка при установке printers.rte"
    exit 1
  else
    exit 0
  fi
}
install_printers "
   next_type                = "n"

#----
# Вопросы: 5,6,8
# Здесь cmd_opt используется как окно "OK/Отмена". Следует также отметить,
# что для просмотра текста опции применяется команда dspmsg. Это
# позволяет преобразовывать сообщения.
# Примечание: значение id_seq_num для опции равно 0. Для каждого заголовка name_hdr
#       разрешена только одна опция, и ее номер id_seq_num должен быть равен 0.
#----
sm_cmd_opt:
      id                        = "demo_queue_opt"
   id_seq_num = "0"
   disc_field_name       = ""
      name                      = "Добавить очередь печати"
   name_msg_file             = "smit.cat"
      name_msg_set              = 52
      name_msg_id               = 41
   op_type = "l"
      cmd_to_list               = "x()\
{
if [ $SMIT = \"a\" ] \n\
then \n\
dspmsg -s 52 smit.cat 56 \
'Для автоматической установки поддержки принтера нажмите Enter.\n\
Для отмены нажмите F3.\n\
'\n\
else \n\
dspmsg -s 52 smit.cat 57 'Для автоматической установки программного обеспечения
принтера щелкните на данном пункте.\n' \n\
fi\n\
} \n\
x"
   entry_type = "t"
      multi_select = "n"

#-------------------------------------------------------------------
#
# Пример 2
# ------
# Цель: Добавление языка для уже установленного приложения. Часто
#       рекомендуется предоставить пользователю некоторую информацию,
#       перед тем как открывать какое-либо окно диалога. Для этой цели
#       можно использовать имена заголовков (sm_name_hdr).
#       В данном примере для определения устанавливаемого языка и
#       установочного устройства используются два заголовка. WBlazer примет
#       В окне диалога предусмотрены поля ввода информации,
#       необходимой для выполнения задачи.
#
# Вопросы:
#       1. Сохранение выходных данных от последовательных name_hdrs
#          в cooked_field_name
#       2. Вызов функций getopts внутри cmd_to_exec для обработки
#          информации в cmd_opt
#       3. Определение кольцевого списка или списка cmd_to_list для просмотра
#          на экране значений cmd_opts
```

```
#------------------------------------------------------------

#----
# Вопрос: 1
# Это первый name_hdr. Он вызывается из menu_opt для данной функции.
# Мы хотим сохранить данные, введенные пользователем, для последующего
# использования в окне диалога. Параметр, передаваемый в cmd_to_classify,
# вводится/выбирается пользователем. Cmd_to_classify очищает
# вывод и сохраняет его в переменной, определяемой в cooked_field_name.
# При этом переопределяется значение, заданное по умолчанию для
# вывода cmd_to_classify, которое равно _cookedname. Переопределение
# необходимо из-за того, что нам нужно сохранить вывод
# из следующего name_hdr.
#----
sm_name_hdr:
    id                          = "demo_mle_inst_lang_hdr"
    next_id                     = "demo_mle_inst_lang"
    option_id                   = "demo_mle_inst_lang_select"
    name                        = "Добавить язык для уже установленного приложения"
    name_msg_file               = "smit.cat"
    name_msg_set                = 53
    name_msg_id                 = 35
    type                        = "j"
    ghost                       = "n"
    cmd_to_classify             = "\
        foo() {
            echo $1 | sed -n \"s/[^[]*\\[\\([^]]*\\).*/\\1/p\"
        }
        foo"
    cooked_field_name           = "add_lang_language"
    next_type                   = "n"
    help_msg_id                 = "2850325"

sm_cmd_opt:
    id                          = "demo_mle_inst_lang_select"
    id_seq_num = "0"
    disc_field_name             = "add_lang_language"
    name                        = "Устанавливаемый язык"
    name_msg_file               = "smit.cat"
    name_msg_set                = 53
    name_msg_id                 = 20
    op_type = "l"
    entry_type                  = "n"
    entry_size                  = 0
    required                    = ""
    prefix                      = "-l "
    cmd_to_list_mode            = "a"
    cmd_to_list                 = "/usr/lib/nls/lsmle -l"
    help_msg_id                 = "2850328"

#----
# Вопрос: 1
# Это второй name_hdr. Здесь данные, введенные пользователем, передаются
# непосредственно через cmd_to_classify и сохраняются в переменной
# add_lang_input.
#----
sm_name_hdr:
    id                          = "demo_mle_inst_lang"
    next_id                     = "demo_dialog_add_lang"
    option_id                   = "demo_add_input_select"
    has_name_select             = "y"
    name                        = "Добавить язык для уже установленного приложения"
    name_msg_file               = "smit.cat"
    name_msg_set                = 53
    name_msg_id                 = 35
    type                        = "j"
    ghost                       = "n"
```

```
    cmd_to_classify           = "\
        foo() {
            echo $1
        }
        foo"
    cooked_field_name         = "add_lang_input"
    next_type = "d"
    help_msg_id               = "2850328"

sm_cmd_opt:
    id                        = "demo_add_input_select"
    id_seq_num = "0"
    disc_field_name           = "add_lang_input"
    name                   = "Устройство/каталог для загрузки программного обеспечения"
    name_msg_file             = "smit.cat"
    name_msg_set              = 53
    name_msg_id               = 11
    op_type = "l"
    entry_type = "t"
    entry_size                = 0
    required                  = "y"
    prefix                    = "-d "
    cmd_to_list_mode          = "1"
    cmd_to_list               = "/usr/lib/instl/sm_inst list_devices"
    help_msg_id               = "2850313"

#----
# Вопрос: 2
# Каждое значение cmd_opts преобразуется к специальному формату
# (тире, затем одиночный символ и необязательный параметр), позволяющий
# обрабатывать его командой getopts. Двоеточие после буквы в команде
# getopts означает, что после опции с тире ожидается параметр.
# Это наилучший способ обработки информации, содержащейся в cmd_opt,
# если задается несколько опций, в особенности если одна из них
# пропускается, что приводит к нарушению последовательности $1, $2,
# и т. д.
#----
sm_cmd_hdr:
    id              = "demo_dialog_add_lang"
    option_id       = "demo_mle_add_app_lang"
    has_name_select        = ""
    name                      = "Добавить язык для уже установленного приложения"
    name_msg_file             = "smit.cat"
    name_msg_set              = 53
    name_msg_id               = 35
    cmd_to_exec     = "\
       foo()
       {
       while getopts d:l:S:X Option \"$@\"
       do
          case $Option in
             d) device=$OPTARG;;
             l) language=$OPTARG;;
             S) software=$OPTARG;;
             X) extend_fs="-X";;
          esac
       завершено

       if [[ `/usr/lib/assist/check_cd -d $device` = '1' ]]
       то
          /usr/lib/assist/mount_cd $device
          CD_MOUNTED=true
       fi

       if [[ $software = \"ALL\" ]]
       то
          echo "Устанавливается весь пакет программ для $language..."
```

```
       else
          echo "Устанавливается $software для $language..."
       fi
       exit $RC
       }
       foo"
    ask                       = "y"
    ghost                     = "n"
    help_msg_id               = "2850325"

sm_cmd_opt:
    id                        = "demo_mle_add_app_lang"
    id_seq_num = "0"
    disc_field_name           = "add_lang_language"
    name                      = "Устанавливаемый язык"
    name_msg_file             = "smit.cat"
    name_msg_set              = 53
    name_msg_id               = 20
    entry_type                = "n"
    entry_size                = 0
    required                  = "y"
    prefix                    = "-l "
    cmd_to_list_mode          = "a"
    help_msg_id               = "2850328"

#----
# Вопрос: 2
# Перед выбираемым пользователем значением стоит поле prefix;
# оба значения (и prefix, и выбранное значение) передаются в
# cmd_to_exec для обработки командой getopts.
#----
sm_cmd_opt:
    id                          = "demo_mle_add_app_lang"
    id_seq_num            = "020"
    disc_field_name             = "add_lang_input"
    name                  = "Устройство/каталог для загрузки программного обеспечения"
    name_msg_file               = "smit.cat"
    name_msg_set                = 53
    name_msg_id                 = 11
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                      = "-d "
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "/usr/lib/instl/sm_inst list_devices"
    help_msg_id                 = "2850313"

sm_cmd_opt:
    id                          = "demo_mle_add_app_lang"
    id_seq_num            = "030"
    name                  = "Установленное приложение"
    name_msg_file               = "smit.cat"
    name_msg_set                = 53
    name_msg_id           = 43
    op_type = "l"
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                = "-S "
    cmd_to_list_mode      = ""
    cmd_to_list           = "\
       list_messages ()
       {
          language=$1
          device=$2
          lslpp -Lqc | cut -f2,3 -d':'
       }
```

```
       list_messages"
    cmd_to_list_postfix   = "add_lang_language add_lang_input"
    multi_select          = ","
    value_index           = 0
    disp_values           = "ALL"
    help_msg_id           = "2850329"

#----
# Вопрос: 3
# Здесь вместо cmd_to_list указывается набор значений кольцевого
# списка в поле disp_values (через запятую). При нажатии пользователем
# клавиши Tab в поле ввода cmd_opt будет появляться по одному элементу из этого
# списка. Однако вместо того, чтобы передавать в cmd_hdr значение "yes" (да)
# или "no" (нет), рекомендуется использовать поле aix_values для передачи значения
# -X, или не передавать никакое значение. Список в поле aix_values должен в точности
# совпадать со списком в поле disp_values.
#----
sm_cmd_opt:
      id_seq_num = "40"
   id                         = "demo_mle_add_app_lang"
   disc_field_name       = ""
      name = "Увеличить объем файловых систем при необходимости?"
   name_msg_file              = "smit.cat"
   name_msg_set               = 53
      name_msg_id = 12
      op_type = "r"
   entry_type                 = "n"
   entry_size                 = 0
   required                   = "y"
      multi_select = "n"
   value_index           = 0
      disp_values = "yes,no"
        values_msg_file = "sm_inst.cat"
      values_msg_set = 1
      values_msg_id = 51
      aix_values = "-X,"
      help_msg_id = "0503005"

#-------------------------------------------------------------------
#
# Пример 3
# ------
# Цель: Показать характеристики логического тома. Имя логического
#   тома вводится пользователем и передается в
#   cmd_hdr как имя  _rawname.
#
# Вопросы:     1. _rawname
#               2. Кольцевой список & aix_values
#-------------------------------------------------------------------

#----
# Вопрос: 1
# Имя rawname не требуется, так как мы используем только один заголовок name_hdr
# и можем применять _rawname в качестве имени переменной по умолчанию.
#----
sm_name_hdr:
      id = "demo_lspv"
      next_id = "demo_lspvd"
      option_id = "demo_cmdlvmpvns"
   has_name_select        = ""
      name = "Оглавление физического тома"
   name_msg_file              = "smit.cat"
      name_msg_set = 15
      name_msg_id = 100
   type                       = "j"
   ghost                   = ""
   cmd_to_classify         = ""
```

```
   raw_field_name          = ""
   cooked_field_name       = ""
   next_type = "d"
      help_msg_id = "0516100"

sm_cmd_opt:
   id_seq_num = "0"
      id = "demo_cmdlvmpvns"
      disc_field_name = "PVName"
      name = "Имя физического тома"
   name_msg_file             = "smit.cat"
      name_msg_set = 15
      name_msg_id = 101
   op_type = "l"
   entry_type = "t"
   entry_size                = 0
      required = "+"
   cmd_to_list_mode          = "1"
      cmd_to_list = "lsvg -o|lsvg -i -p|grep -v '[:P]'| \
            cut -f1 -d' '"
   cmd_to_list_postfix   = ""
      multi_select = "n"
      help_msg_id = "0516021"

#----
# Вопрос: 1
# Значение cmd_to_discover_postfix передает имя физического тома,
# которое представляет собой исходные данные, выбираемые пользователем в
# name_hdr - _rawname.
#----
sm_cmd_hdr:
      id = "demo_lspvd"
      option_id = "demo_cmdlvmlspv"
   has_name_select           = "y"
      name = "Оглавление физического тома"
   name_msg_file             = "smit.cat"
      name_msg_set = 15
      name_msg_id = 100
      cmd_to_exec = "lspv"
      ask = "n"
      cmd_to_discover_postfix = "_rawname"
      help_msg_id = "0516100"

sm_cmd_opt:
      id_seq_num = "01"
      id = "demo_cmdlvmlspv"
      disc_field_name = "_rawname"
      name = "Имя физического тома"
   name_msg_file             = "smit.cat"
      name_msg_set = 15
      name_msg_id = 101
   op_type = "l"
   entry_type = "t"
   entry_size                = 0
      required = "+"
   cmd_to_list_mode          = "1"
      cmd_to_list = "lsvg -o|lsvg -i -p|grep -v '[:P]'| \
            cut -f1 -d' '"
      help_msg_id = "0516021"

#----
# Вопрос: 2
# Здесь кольцевой список из трех значений совпадает со списком aix_values,
# который мы хотим передать в cmd_to_exec для sm_cmd_hdr.
#----
sm_cmd_opt:
      id_seq_num = "02"
```

```
      id = "demo_cmdlvmlspv"
      disc_field_name = "Option"
      name = "Опция просмотра:"
   name_msg_file             = "smit.cat"
      name_msg_set = 15
      name_msg_id = 92
      op_type = "r"
   entry_type                = "n"
   entry_size                = 0
      required = "n"
   value_index           = 0
      disp_values = "status,logical volumes,physical \
                    partitions"
      values_msg_file = "smit.cat"
      values_msg_set = 15
      values_msg_id = 103
      aix_values = "  ,-l,-p"
      help_msg_id = "0516102"

#-------------------------------------------------------------------
#
# Пример 4
# ------
# Цель: Изменить/показать дату & время
#
# Вопросы:      1. Использование заголовка динамического окна для получения
#                  значений переменных для следующего окна диалога.
#               2. Использование cmd_to_discover для задания нескольких
#                  начальных значений cmd_opt.
#               3. Переупорядочение параметров в cmd_to_exec.
#-------------------------------------------------------------------

#----
# Вопрос: 1
# Данный заголовок динамического окна name_hdr получает два значения, которые
# хранятся в переменных daylight_y_n и time_zone и используются в cmd_opts
# следующего окна диалога. Выходные значения cmd_to_classify разделяются двоеточием,
# как и в списке имен полей, определяемом в cooked_field_name.
#----
sm_name_hdr:
      id = "demo_date"
      next_id = "demo_date_dial"
      option_id = "date_sel_opt"
   name_msg_set            = 0
   name_msg_id             = 0
      ghost = "y"
      cmd_to_classify           = "\
if [ $(echo $TZ | awk '{ \
  if (length($1) <=6 ) {printf(\"2\")} \
  else {printf(\"1\")} }') = 1 ] \n\
then\n\
    echo $(dspmsg smit.cat -s 30 18 'yes')\":$TZ\"\n\
else\n\
    echo $(dspmsg smit.cat -s 30 19 'no')\":$TZ\"\n\
fi #"
      cooked_field_name = "daylight_y_n:time_zone"

sm_cmd_opt:
   id_seq_num = "0"
      id = "date_sel_opt"

#----
# Вопросы: 2,3
# Здесь cmd_to_discover получает шесть значений, по одному для каждой
# доступной для изменения опции sm_cmd_opts для данного окна.
# Вывод cmd_to_discover состоит из двух строк; первая начинается с символа #,
# за которым следует список имен переменных, а вторая содержит список их значений. И то, и другое
```

```
# В обоих списках в качестве разделителя используется двоеточие. Следует отметить,
# что cmd_to_exec получает параметры из cmd_opts и переупорядочивает их
# при вызове команды.
#----
sm_cmd_hdr:
      id = "demo_date_dial"
      option_id = "demo_chtz_opts"
   has_name_select           = "y"
      name = "Изменить / Показать дату и время"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 21
      cmd_to_exec = "date_proc () \
# ММ дд чч мм сс гг\n\
# порядок параметров диалога   #  3  4  5  6  7  2\n\
{\n\
date \"$3$4$5$6.$7$2\"\n\
}\n\
date_proc "
      exec_mode = "P"
      cmd_to_discover = "disc_proc() \n\
{\n\
TZ=\"$1\"\n\
echo '#cur_month:cur_day:cur_hour:cur_min:cur_sec:cur_year'\n\
date +%m:%d:%H:%M:%S:%y\n\
}\n\
disc_proc"
   cmd_to_discover_postfix = ""
      help_msg_id = "055101"

#----
# Первые два cmd_opts получают начальные значения
# (disc_field_name) из name_hdr.
#----
sm_cmd_opt:
      id_seq_num = "04"
      id = "demo_chtz_opts"
      disc_field_name = "time_zone"
      name = "Часовой пояс"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 16
   required                  = "y"

sm_cmd_opt:
      id_seq_num = "08"
      id = "demo_chtz_opts"
      disc_field_name = "daylight_y_n"
      name = "Предусмотрен ли в данном часовом поясе переход на летнее время?\n"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 17
   entry_size                = 0

#----
# Последние шесть six cmd_opts получают свои значения из
# cmd_to_discover.
#----
sm_cmd_opt:
      id_seq_num = "10"
      id = "demo_chtz_opts"
      disc_field_name = "cur_year"
      name = "Год (00-99)"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 10
      entry_type = "#"
```

```
      entry_size = 2
      required = "+"
      help_msg_id = "055102"

sm_cmd_opt:
      id_seq_num = "20"
      id = "demo_chtz_opts"
      disc_field_name = "cur_month"
      name = "Месяц (01-12)"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
   name_msg_id               = 11
      entry_type = "#"
      entry_size = 2
      required = "+"
      help_msg_id = "055132"

sm_cmd_opt:
      id_seq_num = "30"
      id = "demo_chtz_opts"
      disc_field_name = "cur_day"
      name = "День (01-31)\n"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 12
      entry_type = "#"
      entry_size = 2
      required = "+"
      help_msg_id = "055133"

sm_cmd_opt:
      id_seq_num = "40"
      id = "demo_chtz_opts"
      disc_field_name = "cur_hour"
      name = "Час (00-23)"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 13
      entry_type = "#"
      entry_size = 2
      required = "+"
      help_msg_id = "055134"

sm_cmd_opt:
      id_seq_num = "50"
      id = "demo_chtz_opts"
      disc_field_name = "cur_min"
      name = "Минута (00-59)"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 14
      entry_type = "#"
      entry_size = 2
      required = "+"
      help_msg_id = "055135"

sm_cmd_opt:
      id_seq_num = "60"
      id = "demo_chtz_opts"
      disc_field_name = "cur_sec"
      name = "Секунда (00-59)"
   name_msg_file             = "smit.cat"
      name_msg_set = 30
      name_msg_id = 15
```

```
entry_type = "#"
entry_size = 2
required = "+"
help_msg_id = "055136"
```

Контроллер системных ресурсов

В этом разделе описан Контроллер системных ресурсов (SRC), предназначенный для управления сложными подсистемами.

SRC - это программа для управления подсистемами. SRC может применяться при разработке подсистем, управляющих одним или несколькими демонами, для создания интерфейса управления системой. SRC предоставляет набор команд для запуска, завершения, трассировки, обновления и получения информации о состоянии подсистемы.

Кроме того, в SRC предусмотрена функция уведомления об ошибках. Она позволяет применять собственные методы исправления ошибок. На тип информации об исправлении не накладывается практически никаких ограничений: метод уведомления должен быть строкой файла, содержащей имя исполняемой программы.

Дополнительная информация о требованиях, которые накладываются на программы SRC, приведена в следующих разделах:

Взаимодействие подсистем с SRC

В SRC подсистемой называется программа или набор связанных программ, которые были совместно разработаны для решения конкретной задачи. Дополнительная информация о свойствах подсистемы приведена в разделе "Контроллер системных ресурсов" книги *Управление операционной системой и устройствами*.

Субсервер - это процесс, которым управляет подсистема и который входит в ее состав.

SRC работает с объектами из класса объектов SRC. При этом подсистемы определяются в SRC как объекты подсистем, а субсерверы - как объекты типа субсервер. Структура, связанная с объектом каждого типа, описана в файле **usr/include/sys/srcobj.h**.

В SRC предусмотрены команды для работы с объектами подсистем, субсерверов, а также группами подсистем. Группой подсистем называется группа любых пользовательских подсистем. Объединение подсистем в группы позволяет выполнить одну команду сразу для нескольких подсистем. Кроме того, для группы подсистем можно создать общий метод уведомления об ошибках.

SRC обменивается данными с подсистемами путем отправки сигналов и обмена пакетами с запросами и ответами. Помимо сигналов SRC поддерживает обмен данными через сокеты и очереди сообщений IPC. Для описания обмена данными между подсистемами и SRC в API SRC предусмотрено несколько функций. Кроме того, в API SRC предусмотрены функции для обмена данными между клиентскими программами и SRC.

SRC и команда init

При работе SRC не используется команда **init**. Тем не менее, SRC расширяет возможности этой команды, предназначенной для порождения процессов. Помимо команд запуска, завершения, трассировки, обновления и получения информации о состоянии подсистем, SRC позволяет управлять работой отдельных подсистем, поддерживает удаленное управление системой и функцию ведения протоколов подсистем.

SRC вызывает команду **init** только при добавлении главного демона SRC, **srcmstr**, в файл **inittab**. По умолчанию демон **srcmstr** указан в файле **inittab**. В этом случае команда **init** запускает демон **srcmstr** во время запуска системы, как и любой другой процесс. Права на запуск демона **srcmstr** есть только у системного администратора и членов системной группы.

Компиляция программ, взаимодействующих с демоном srcmstr

Для того чтобы программа могла работать с демоном **srcmstr**, в ней нужно указать файл **/usr/include/spc.h**, а во время компиляции подключить библиотеку **libsrc.a**. Для программ, обменивающихся данными с SRC путем сигналов, этого делать не нужно.

Операции SRC

Для работы с SRC подсистема должна обмениваться данными с демоном **srcmstr** двумя способами:

- С помощью объекта подсистемы, созданного в классе объектов подсистем SRC.
- С помощью функций SRC, если подсистема не использует механизм сигналов. Эти функции предназначены для отправки ответов на запросы SRC и применяются в случае использования очередей сообщений или сокетов.

Все подсистемы SRC должны поддерживать команду **stopsrc**. Эта команда применяется SRC для завершения работы подсистем и их субсерверов с помощью сигналов **SIGNORM** (обычное завершение), **SIGFORCE** (принудительное завершение) или **SIGCANCEL** (отмена).

Кроме того, подсистема может поддерживать команды **startsrc**, **lssrc-l**, **traceson**, **tracesoff** и **refresh**, функцию отправки подробного отчета о своем состоянии и отчета о состоянии субсервера, а также функцию уведомления об ошибках SRC.

Функции SRC

Для поддержки создания подсистем в SRC предусмотрены следующие функции:

- Общий интерфейс команд для запуска, завершения работы и отправки запросов подсистемам
- Возможность централизованного управления подсистемами и группами подсистем
- Общий формат запросов к подсистемам
- Определение субсервера, однозначно идентифицирующее субсервер в подсистеме
- Возможность определить для подсистемы особые методы уведомления об ошибках
- Возможность определить для подсистемы особые ответы на запросы о состоянии, трассировке и обновлении конфигурации
- Централизованная обработка запросов к подсистемам в сетевой среде

Информация, связанная с данной:

mknotify

mkserver

mksys

refresh

rmnotify

srcmstr

tracesoff

traceson

spc.h

srcobj.h

Объекты SRC

Контроллер системных ресурсов (SRC) работает с тремя классами объектов:

Все функции SRC предназначены для работы с объектами этих классов. Предопределенный набор дескрипторов классов объектов определяет возможный набор конфигураций подсистемы, поддерживаемых SRC.

> **Примечание:** Для работы SRC требуется только класс объектов подсистем. Классы объектов типа субсервера и объектов уведомления необходимы только в некоторых системах.

Класс объектов подсистем

Класс объектов подсистем содержит дескрипторы всех подсистем SRC. Для того чтобы SRC смог работать с подсистемой, ее нужно определить в этом классе.

Дескрипторы класса объектов подсистем определены в структуре **SRCsubsys**, хранящейся в файле **/usr/include/sys/srcobj.h**. Краткое описание дескрипторов подсистем и связанных с ними флагов команд **mkssys** и **chssys** приведено в таблице Дескрипторы объектов подсистем и значения по умолчанию.

Таблица 82. Дескрипторы объектов подсистем и значения по умолчанию

Дескрипторы	Значения по умолчанию	Флаги
Имя подсистемы		-s
Имя команды подсистемы		-p
Аргументы команды		-a
Приоритет выполнения	20	-E
Несколько экземпляров	NO	-Q -q
ИД пользователя		-u
Синоним (ключ)		-t
Запуск	ONCE	-O -R
stdin	/dev/console	-i
stdout	/dev/console	-o
stderr	/dev/console	-e
Способ обмена данными	Сокеты	-K -I -S
Тип сообщений подсистемы		-m
Ключ очереди IPC для обмена данными		-l
Имя группы		-G
Сигнал **SIGNORM**		-n
Сигнал **SIGFORCE**		-f
Показать	Yes	-D -d
Время ожидания	20 секунд	-w
ИД контроля		

Дескрипторы объектов подсистем определяются следующим образом:

Дескрипторы объектов	Определение
Имя подсистемы	Задает имя объекта подсистемы. Размер имени не должен превышать 30 байт, с учетом NULL-терминатора (29 однобайтовых символов или 14 многобайтовых символов). Этот дескриптор должен быть задан в соответствии со стандартом POSIX. Это обязательное поле.
Имя команды подсистемы	Задает полное имя программы, выполняемой командой запуска подсистемы. Размер полного имени не должен превышать 200 байт (199 однобайтовых символов или 99 многобайтовых символов). Имя должно быть задано в соответствии со стандартом POSIX. Это обязательное поле.

Дескрипторы объектов	Определение
Аргументы команды	Задает аргументы команды запуска подсистемы. Размер аргументов не должен превышать 200 байт, с учетом NULL-терминатора (199 однобайтовых символов или 99 многобайтовых символов). Список аргументов анализируется демоном **srcmstr** по тем же правилам, которые применяются оболочками. Например, строки, заключенные в кавычки, передаются как строковый аргумент, а пробелы между строками считаются разделителями аргументов.
Приоритет выполнения	Задает приоритет, с которым будет выполняться процесс подсистемы. Этот приоритет назначается всем подсистемам, запущенным с помощью демона **srcmstr**. Значение по умолчанию - 20.
Несколько экземпляров	Задает число экземпляров подсистемы, которые могут выполняться одновременно. Значение Нет (флаг **-Q**) указывает, что в каждый момент времени может выполняться только один экземпляр подсистемы. В этом случае запрос на запуск еще одного экземпляра подсистемы или подсистемы с тем же ключом очереди сообщений IPC не будет выполнен. Значение Да (флаг **-q**) указывает, что с одной и той же очередью сообщений IPC могут работать несколько подсистем, и одновременно может выполняться несколько экземпляров подсистемы. Значение по умолчанию - Нет.
ИД пользователя	Задает ИД пользователя (в численном виде), у которого есть права на запуск подсистемы. Значение 0 соответствует пользователю root. Это обязательное поле.
Синоним	Задает символьную строку, которая будет применяться в качестве альтернативного имени подсистемы. Размер символьной строки не должен превышать 30 байт, с учетом NULL-терминатора (29 однобайтовых символов или 14 многобайтовых символов). Это необязательное поле.
Запуск	Указывает, должен ли демон **srcmstr** перезапускать подсистему после аварийного завершения работы. Значение RESPAWN (флаг **-R**) указывает, что демон **srcmstr** должен перезапускать подсистему. Значение ONCE (флаг **-O**) указывает, что демон **srcmstr** не должен перезапускать систему, в которой произошел сбой. Если указано значение RESPAWN, то система будет перезапускаться максимум два раза в течение указанного периода ожидания. Если подсистему, в которой произошел сбой, не удалось запустить снова, то запускается метод уведомления. Значение по умолчанию - ONCE.
Стандартный файл/устройство ввода	Задает файл или устройство, из которого система получает ввод. Значение по умолчанию - **/dev/console**. Размер этого поля не должен превышать 200 байт, с учетом NULL-терминатора (199 однобайтовых символов или 99 многобайтовых символов). Если в качестве способа обмена данными выбраны сокеты, то значение этого поля игнорируется.
Стандартный файл/устройство вывода	Задает файл или устройство, на которое система отправляет вывод. Размер этого поля не должен превышать 200 байт, с учетом NULL-терминатора (199 однобайтовых символов или 99 многобайтовых символов). Значение по умолчанию - **/dev/console**.
Стандартный файл/устройство ошибок	Задает файл или устройство, на которое система будет отправлять сообщения об ошибках. Размер этого поля не должен превышать 200 байт, с учетом NULL-терминатора (199 однобайтовых символов или 99 многобайтовых символов). Все ошибки должны обрабатываться методом уведомления. Значение по умолчанию - **/dev/console**. **Примечание:** Информация о серьезных ошибках записывается в протокол ошибок.
Способ обмена данными	Задает способ обмена данными между демоном **srcmstr** и подсистемой. Можно выбрать один из трех способов: IPC (**-I**), сокеты (**-K**) или сигналы (**-S**). Значение по умолчанию - сокеты.
Ключ очереди IPC для обмена данными	Задает десятичное значение, соответствующее ключу очереди IPC. Это значение демон **srcmstr** применяет для обмена данными с подсистемой. Это поле применяется только подсистемами, для которых в качестве способа обмена данными выбрана очередь IPC. Для того чтобы создать уникальный ключ, воспользуйтесь функцией **ftok**, указав полное имя и ИД. Демон **srcmstr** создает очередь сообщений перед запуском подсистемы.
Имя группы	Задает имя группы подсистемы. Размер этого поля не должен превышать 30 байт (29 однобайтовых символов или 14 многобайтовых символов). Это необязательное поле.
Тип сообщений подсистемы	Задает тип сообщений, которые будут отправляться в очередь сообщений подсистемы. Подсистема применяет это значение при получении сообщений с помощью функции **msgrcv** или **msgxrcv**. Это поле является обязательным, если в качестве способа обмена данными выбрана очередь сообщений.
Сигнал SIGNORM	Задает значение, которое должно быть передано подсистеме при отправке запроса на обычное завершение. Это значение применяется только системами, которые в качестве способа обмена данными используют сигналы.
Сигнал SIGFORCE	Задает значение, которое должно передаваться подсистеме при отправке запроса на принудительное завершение. Это значение применяется только системами, которые в качестве способа обмена данными используют сигналы.
Показывать значение	Указывает, должно ли в выводе команды **lssrc -a** и **lssrc -g** указываться состояние неработающей подсистемы. Если оно должно быть показано, то укажите флаг **-d**, если нет - флаг **-D**. Значение по умолчанию - **-d** (показывать).

Дескрипторы объектов	Определение
Время ожидания	Задает интервал времени в секундах, в течение которого подсистема должна запуститься или завершить работу перед тем, как будет выполнено альтернативное действие. Значение по умолчанию - 20 секунд.
ИД контроля	Задает ИД контроля для подсистемы. ИД создается автоматически демоном **srcmstr** при определении подсистемы и используется средством защиты системы, если оно настроено. Значение этого поля нельзя задать или изменить с помощью программы.

Класс объектов типов субсервера

Если в подсистеме есть субсерверы, которым демон **srcmstr** будет отправлять команды уровня субсервера, то в данном классе должен быть соответствующий объект.

Этот класс объектов содержит три дескриптора, которые определены в структуре **SRCsubsvr**, хранящейся в файле **srcobj.h**:

Дескриптор	Определение
ИД субсервера (ключ)	Задает имя объекта типа субсервера. Набор имен для типов субсерверов совпадает с набором допустимых значений флага **-t** команд субсервера. Длина имени не должна превышать 30 байт, с учетом NULL-терминатора (29 однобайтовых символов или 14 многобайтовых символов).
Имя подсистемы-владельца	Задает имя подсистемы, которой принадлежит объект субсервера. В этом поле указывается ссылка на класс объектов подсистем SRC.
Кодовый знак	Задает десятичное число, идентифицирующее субсервер. Кодовый знак передается подсистеме, управляющей субсервером, в поле `object` структуры **subreq**, то есть структуры запроса SRC. Если в команде указано и имя объекта субсервера, демон **srcmstr** передает подсистеме кодовый знак в поле `objname` структуры **subreq**. См. "Пример структуры запроса SRC" в файле **spc.h**.

В командах субсервер идентифицируется по типу необязательному имени экземпляра. С помощью типа субсервера демон SRC определяет подсистему, управляющую субсервером; имя субсервера не применяется.

Класс объектов уведомления

С помощью этого класса демон **srcmstr** может вызвать функции подсистемы для обработки ошибок в ее работе. Когда демон SRC получает сигнал **SIGCHLD**, означающий завершение процесса подсистемы, он проверяет состояние подсистемы (обслуживаемой демоном **srcmstr**) и определяет, связано ли завершение работы с вызовом команды **stopsrc**. Если команда **stopsrc** не выполнялась, завершение считается аварийным. Если в определении не задана опция перезапуска, либо попытка перезапуска не удалась, демон **srcmstr** пытается прочитать объект уведомления, связанный с данной подсистемой. Если такой объект задан, то выполняется метод, связанный с подсистемой.

Если объект подсистемы не найден в классе объектов уведомления, демон **srcmstr** проверяет, входит ли подсистема в группу. Если да, демон **srcmstr** пытается найти в классе объектов уведомления объект группы. Если такой объект есть, то вызывается связанный с ним метод. Таким образом, группа подсистем может применять общий метод.

> **Примечание:** Метод уведомления подсистемы считается более приоритетным, чем метод уведомления группы. За счет этого в подсистеме, относящейся к группе одновременно запускаемых подсистем, может быть создан собственный метод восстановления и очистки.

Объекты уведомления содержат два дескриптора:

Дескриптор	Определение
Имя подсистемы или **Имя группы**	Задает имя подсистемы или группы, для которой определяется метод уведомления.
Метод уведомления	Задает путь к процедуре, которая должна выполняться демоном **srcmstr** при аварийном завершении работы системы или группы.

Уведомление об аварийном завершении полезно в том случае, когда перед перезапуском подсистемы требуется выполнить специальные действия по восстановлению и очистке. Кроме того, это позволяет собирать информацию для последующего анализа причины аварийного завершения работы подсистемы.

Для создания объектов уведомления предназначена команда **mknotify**. Для изменения метода уведомления нужно удалить текущий объект уведомления с помощью команды **rmnotify** и создать новый объект.

Команда	Определение
mknotify	Добавляет метод уведомления в базу данных конфигурации SRC
rmnotify	Удаляет метод уведомления из базы данных конфигурации SRC

Демон **srcmstr** заносит в протокол информацию о действиях подсистемы по исправлению ошибок. Ответственность за отправку сообщений об ошибках ложится на саму подсистему.

Типы обмена данными SRC

Контроллер системных ресурсов (SRC) поддерживает три способа обмена данными: сигналы, сокеты и очереди сообщений IPC.

Выбранный способ обмена данными определяет набор функций SRC, доступный подсистеме.

Примечание: Все подсистемы, независимо от способа обмена данными, указанного в объекте подсистемы, должны поддерживать обмен ограниченным набором сигналов. В подсистеме должна быть определена функция для обработки сигнала **SIGTERM** (завершение и отмена). При получении сигнала **SIGTERM** подсистема должна освободить все ресурсы и завершить свою работу.

Дополнительная информация о способах обмена данными с SRC приведена в следующих разделах:

В таблице Обмен данными между демоном srcmstr и подсистемами описаны все способы обмена данными и связанные с ними функции SRC.

Функция	**С помощью IPC или сокетов**	**С помощью сигналов**
start		
subsystem	SRC порождает процесс подсистемы.	SRC порождает процесс подсистемы.
subserver	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается
обычное завершение		
подсистема	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Подсистеме отправляется сигнал **SIGNORM**.
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
принудительное завершение		
подсистема	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Подсистеме отправляется сигнал **SIGFORCE**
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
завершение и отмена		
подсистема	Группе процессов подсистемы отправляется сигнал **SIGTERM**, а затем - **SIGKILL**.	Группе процессов подсистемы отправляется сигнал **SIGTERM**, а затем - **SIGKILL**.

Функция	С помощью IPC или сокетов	С помощью сигналов
получить краткую информацию о состоянии		
подсистема	Реализуется SRC (нет соответствующего запроса к подсистеме).	Реализуется SRC (нет соответствующего запроса к подсистеме).
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
получить подробную информацию о состоянии		
подсистема	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
включить/выключить трассировку		
подсистема	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
refresh		
подсистема	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
субсервер	Подсистеме отправляется запрос через очередь сообщений IPC или сокет.	Не поддерживается.
notify		
подсистема	Реализуется с помощью метода подсистемы.	Реализуется с помощью метода подсистемы.

Обмен сигналами

Обмен сигналами - это основной способ обмена данными между демоном **srcmstr** и подсистемой. Поскольку сигналы передаются только в одном направлении, подсистемы, использующие механизм сигналов, распознают только запрос на завершение работы. Они не распознают запросы на получение подробной информации о состоянии, обновление и трассировку. Кроме того, они не распознают субсерверы.

Для работы с механизмом сигналов в подсистеме должна быть реализована процедура обработки запросов **SIGNORM** и **SIGFORCE**, например **sigaction**, **sigvec** или **signal**.

Подсистемы, использующие механизм сигналов, должны быть описаны в классе объектов подсистем SRC с помощью команды **mkssys -Snf**, функции **defssys** или **addssys**.

Элемент	Дескриптор
addssys	Добавляет определение подсистемы в базу данных конфигурации SRC
defssys	Инициализирует новое определение подсистемы значениями по умолчанию
mkssys	Добавляет определение подсистемы в базу данных конфигурации SRC

Обмен данными через сокеты

Сокеты представляют собой другой способ обмена данными, который может применяться в подсистемах. Этот тип применяется демоном **srcmstr** по умолчанию. Дополнительная информация приведена в разделе Обзор сокетов*Communications Programming Concepts*.

Демон **srcmstr** получает через сокеты запросы на обработку от процесса. Если вы выберете этот способ обмена данными, демон **srcmstr** создаст сокет подсистемы, через который она будет получать запросы демона **srcmstr**. Для локальных подсистем создаются сокеты UNIX (**AF_UNIX**). Для удаленных подсистем создаются сокеты Internet (**AF_INET**). Ниже описана последовательность обработки команды:

1. Процесс команды получает команду от устройства ввода, создает запрос на обработку и отправляет дейтаграмму UDP с запросом на обработку демону **srcmstr** через стандартный порт SRC. Сокеты **AF_INET** описаны в файле **/etc/services**.
2. Демон **srcmstr** получает запрос на обработку через стандартный порт SRC. После этого он считывает исходный адрес системы из структуры **sockaddr** функции **socket** и добавляет этот адрес и номер порта к запросу на обработку.
3. Демон **srcmstr** вызывает функции **srcrrqs** и **srcsrpy**. Если он может выполнить запрос, то выполняется обработка, результат которой возвращается процессу команды. Остальные запросы передаются соответствующей подсистеме через порт, указанный в запросе на обработку.
4. Подсистема работает с портом, выделенным ей ранее демоном **srcmstr**. (Порт выделяется подсистеме демоном **srcmstr** во время ее запуска.) Подсистема выполняет запрос на обработку и отправляет ответ процессу команды.
5. Процесс команды получает ответ через указанный порт.

Права доступа к файлу и адреса сокетов, применяемые демоном **srcmstr**, хранятся во временных каталогах **/dev/SRC** и **/dev/.SRC-unix**. Информация в этих каталогах предназначена только для SRC, хотя ее можно просмотреть с помощью команды **ls**.

Очереди сообщений и сокеты предоставляют одни и те же функции для обмена данными с подсистемами.

Элемент	Дескриптор
srcrrqs	Сохраняет целевой адрес ответа подсистемы на полученный пакет. (См. также процедуру **srcrrqs_r** с защитой нитей)
srcsrpy	Отправляет пакет с ответом подсистемы на полученный запрос.

Обмен данными с помощью очередей сообщений IPC

Очереди сообщений IPC выполняют те же функции, что и сокеты. Оба способа обмена данными позволяют работать со всеми функциями SRC.

При обмене данными через очередь сообщений IPC демон **srcmstr** получает запросы на обработку от процесса команды через сокет, а затем отправляет сообщение SRC для подсистемы в очередь сообщений IPC. Очередь сообщений создается при запуске подсистемы. Ниже описана последовательность действий, которая выполняется при обмене данными подсистемы с демоном **srcmstr** через очередь сообщений:

1. Демон **srcmstr** считывает ИД очереди сообщений из объекта подсистемы SRC и отправляет сообщение подсистеме.
2. Подсистема получает команду из очереди сообщений с помощью функции **msgrcv** в виде структуры **subreq**, содержащей запрос к подсистеме.
3. Подсистема получает ИД тега, который указывается в ответе на сообщение, с помощью функции **srcrrqs**.
4. Подсистема интерпретирует и обрабатывает полученную команду. В зависимости от команды, подсистема размещает ответ в структуре данных **svrreply** или **statcode**. Дополнительная информация об этих структурах приведена в файле **/usr/include/spc.h**.
5. Подсистема отправляет ответ процессу команды с помощью функции **srcsrpy**.

Создание подсистем, взаимодействующих с SRC

Команды Контроллера системных ресурсов (SRC) - это исполняемые команды, которые получают аргументы из командной строки.

После проверки синтаксиса команды вызываются динамические функции SRC для создания дейтаграммы UDP и ее отправки демону **srcmstr**.

Дополнительная информация о функциях SRC и их использовании подсистемами для связи с главным процессом SRC приведена в следующих разделах:

Получение подсистемами запросов SRC

Способ получения подсистемой запросов от SRC зависит от выбранного способа обмена данными. Демон **srcmstr** получает запросы на обработку от процесса команды через сокеты и создает необходимый сокет или очередь сообщений для передачи этого запроса. Подсистема должна проверить, что сокет или очередь сообщений действительно были созданы. Рекомендации по программированию пакетов с запросами SRC для подсистем, применяющих различные способы обмена данными, приведены в следующих разделах:

Примечание: Во всех подсистемах, независимо от применяемого способа обмена данными, должна быть определена процедура обработки сигнала **SIGTERM**.

Получение сигналов SRC

В подсистемах, обменивающихся с SRC сигналами, должна быть определена процедура обработки сигналов **SIGNORM** и **SIGFORCE**. В ней может использоваться любой метод обработки сигналов. Ниже приведено два примера подобных функций.

Функция	Описание
Функции **sigaction**, **sigvec** и **signal**	Задают действие, которое должно быть выполнено при получении сигнала.
Функции **sigset**, **sighold**, **sigrelse** и **sigignore**	Предоставляют для прикладных процессов расширенные функции для работы с сигналами и управления ими.

Получение пакетов с запросами SRC через сокеты

Ниже приведены рекомендации по созданию подсистем, получающих пакеты с запросами SRC через сокеты:

- Включите в программу подсистемы файл **/usr/include/spc.h**, содержащий структуру подсистемы SRC. Эта структура применяется подсистемой для ответа на команды SRC. Кроме того, файл **spc.h** содержит файл **srcerrno.h**, который не нужно указывать особо. Файл **srcerrno.h** содержит определения ошибок, необходимые для поддержки демона.
- При запуске подсистемы, применяющей сокеты, дескриптор файла 0 присваивается сокету, через который подсистема получает пакеты с запросами SRC. Подсистема должна убедиться, что этот дескриптор действительно был назначен сокету с помощью функции **getsockname**, которая возвращает адрес сокета подсистемы. Если дескриптор сокета отличен от 0, подсистема должна занести в протокол сообщение об ошибке и завершить свою работу. Информация о том, как получить адрес сокета подсистемы с помощью функции **getsockname** приведена в разделе "Пример программы чтения дейтаграммы Internet" в книге *Communications Programming Concepts*.
- Если подсистема работает с несколькими сокетами, то с помощью функции **select** она может определить, в каком из сокетов есть данные. Дополнительная информация по выполнению этой задачи с помощью функции **select** приведена в разделе "Пример программы проверки ожидающих соединений" в книге *Communications Programming Concepts*.
- Для получения пакета с запросом из сокета предназначена функция **recvfrom**.

  > **Примечание:** Целевой адрес для возврата ответа подсистемы хранится в полученном пакете с запросом SRC. Его не следует путать с адресом, который функция **recvfrom** возвращает в качестве одного из своих параметров.

  После получения пакета с помощью функции **recvfrom** можно вызвать функцию **srcrrqs**, которая возвращает указатель на статическую структуру **srchdr**. Этот указатель позволяет получить адрес, по которому нужно отправить ответ подсистемы. Эта структура переопределяется при каждом вызове функции **srcrrqs**, поэтому ее нужно сохранить в некоторой переменной, если она еще потребуется после следующего вызова **srcrrqs**.

Получение пакетов с запросами SRC с помощью очереди сообщений

Ниже приведены рекомендации по созданию подсистем, получающих пакеты с запросами SRC из очередей сообщений:

- Включите в программу подсистемы файл **/usr/include/spc.h**, содержащий структуру подсистемы SRC. Эта структура применяется подсистемой для ответа на команды SRC. Кроме того, файл **spc.h** содержит файл **srcerrno.h**, который не нужно указывать особо. Файл **srcerrno.h** содержит определения ошибок, необходимые для поддержки демона.
- Укажите при компиляции опцию **-DSRCBYQUEUE**. В результате поле типа сообщения (`mtype`*)* станет первым в структуре **srcreq**. Эта структура применяется при получении пакетов SRC.
- После запуска подсистемы необходимо выполнить функцию **msgget** и проверить, что во время запуска была создана очередь сообщений. Если она не была создана, подсистема должна занести в протокол сообщение об ошибке и завершить работу.
- Если подсистема работает с несколькими очередями сообщений, то с помощью функции **select** она может определить, в какой из очередей есть данные. Дополнительная информация по выполнению этой задачи с помощью функции **select** приведена в разделе "Пример программы проверки ожидающих соединений" в книге *Communications Programming Concepts*.
- Для получения пакета из очереди сообщений предназначены функции **msgrcv** и **msgxrcv**. Адрес, по которому подсистема должна отправить пакет с ответом, указан в полученном пакете.
- После получения пакета с помощью функции **msgrcv** или **msgxrcv** вызовите функцию **srcrrqs**. Эта функция возвращает указатель на статическую структуру **srchdr**, которая переопределяется при каждом вызове функции **srcrrqs**. Этот указатель позволяет получить адрес, по которому нужно отправить ответ подсистемы.

Обработка подсистемами пакетов с запросами SRC

В подсистемах должна быть определена реакция на получение запроса на остановку. Помимо этого, подсистемы могут поддерживать запросы на запуск, получение информации о состоянии, трассировку и обновление.

Обработка пакета с запросом разбивается на два этапа:

Чтение пакета с запросом SRC

Подсистема получает пакет с запросом SRC в виде структуры **srcreq**, описанной в файле **/usr/include/spc.h**. Запрос подсистемы расположен в структуре **subreq**, вложенной в структуру **srcreq**:

```
struct subreq
      short object;         /*объект для обработки*/
      short action;         /*действие START, STOP, STATUS, TRACE,\
                             REFRESH*/
      short parm1; /*зарезервировано для переменных*/
      short parm2; /*зарезервировано для переменных*/
      char objname; /*имя объекта*/
```

Поле `object` структуры **subreq** задает объект, над которым нужно выполнить действие. Если запрос относится к подсистеме, в поле `object` указывается константа SUBSYSTEM. В противном случае в поле `object` указывается кодовый знак, либо в поле `objname` указывается PID субсервера в виде символьной строки. Ответственность за правильный выбор объекта, к которому относится запрос, ложится на подсистему.

Поле `action` задает запрошенное действие над подсистемой. Все подсистемы должны поддерживать действия START, STOP и STATUS. При необходимости, могут поддерживаться действия TRACE и REFRESH.

Поля `parm1` и `parm2` применяются для различных целей, в зависимости от запрошенного действия.

Действие	parm1	parm2
STOP	NORMAL или FORCE	
STATUS	LONGSTAT или SHORTSTAT	
TRACE	LONGTRACE или SHORT-TRACE	TRACEON или TRACEOFF

Для действий START и REFRESH поля `parm1` и `parm2` не используются.

Ответ подсистемы на запрос SRC

Действия, которые выполняет система в ответ на большинство запросов SRC, задаются при определении объекта подсистемы в SRC. Структуры, применяемые подсистемами при ответе на запросы SRC, определены в файле **/usr/include/spc.h**. Для обработки команд в подсистемах могут применяться следующие динамические функции SRC:

Функция	Описание
srcrrqs	Позволяет подсистеме сохранить заголовок запроса.
srcsrpy	Позволяет подсистеме отправить ответ на запрос.

Для обработки запросов на получение информации о состоянии требуется выполнить несколько задач и функций.

В ответ на неизвестный или неверный запрос подсистема должна отправить пакет с этим запросом и кодом ошибки **SRC_SUBICMD**. Коды действий 0-255 зарезервированы в SRC для внутреннего использования. При получении подсистемой запроса с неправильным кодом действия она должна вернуть код ошибки **SRC_SUBICMD**. Правильные коды действий, поддерживаемые SRC, перечислены в файле **spc.h**. Кроме того, вы можете определить собственные коды действий. Код действия считается неправильным, если он не определен в SRC и подсистеме.

> **Примечание:** Коды действий 0-255 зарезервированы для SRC.

Обработка запросов SRC на получение информации о состоянии

Подсистема может получить запрос на отправку отчета о состоянии одного из трех типов: подробного отчета о состоянии подсистемы, краткого отчета о состоянии субсервера и подробного отчета о состоянии субсервера.

> **Примечание:** Краткие отчеты о состоянии подсистемы составляются демоном **srcmstr**. Код состояния и константы, описывающие состояние, которые указываются в отчетах этого типа, определены в файле **/usr/include/spc.h**. В таблице Константы состояния перечислены обязательные и дополнительные коды состояния, которые могут быть возвращены в ответ на запрос.

Коды состояния, которые могут встретиться в ответе

Значение	Описание	Подсистема	Субсервер
SRCWARN	Получен запрос на остановку. (Будет остановлена в течение 20 секунд.)	X	X
SRCACT	Запущена и работает.	X	X
SRCINAC	Не работает.		
SRCINOP	Неисправна.	X	X
SRCLOSD	Закрыта.		
SRCLSPN	Закрывается.		
SRCNOSTAT	Простаивает.		

Значение	Описание	Подсистема	Субсервер
SRCOBIN	Открыта, но не работает.		
SRCOPND	Открыта.		
SRCOPPN	Открывается.		
SRCSTAR	Запускается.		X
SRCSTPG	Останавливается.	X	X
SRCTST	Выполняется проверка.		
SRCTSTPN	Ожидание проверки.		

Команда SRC **lssrc** записывает полученную информацию в стандартный вывод. Информация, которая возвращается в ответ на запрос о подробном отчете о состоянии, определяется программистом подсистемы. При необходимости подсистема может отслеживать состояние своих субсерверов и сообщать о его изменении. С помощью функции **srcstathdr** можно получить стандартный заголовок отчета о состоянии и добавить его к своим данным о состоянии.

Обработку запроса на получение информации о состоянии рекомендуется разбить на следующие шаги:

1. Для возврата отчета о состоянии подсистемы (краткого или подробного), создайте массив структур **statcode** и структуру **srchdr**. Структура **srchdr** должна располагаться в начале буфера, отправляемого в ответ на запрос о состоянии. Структура **statcode** определена в файле **/usr/include/spc.h**.

   ```
   struct statcode
   {
      short objtype;
      short status;
      char objtext [65];
      char objname [30];
   };
   ```

2. Укажите в поле `objtype` константу SUBSYSTEM, если информация о состоянии относится к подсистеме, или кодовый знак субсервера, если информация относится к субсерверу.
3. Укажите в поле `status` одну из констант состояния SRC, перечисленных в файле **spc.h**.
4. Укажите в поле `objtext` текст NLS, описывающий состояние. Строка, указанная в этом поле, должна заканчиваться символом NULL.
5. Укажите в поле `objname` имя подсистемы или субсервера, к которому относится значение `objtext`. Строка, указанная в этом поле, должна заканчиваться символом NULL.

   Примечание: Подсистема и инициатор запроса могут договориться об отправке другой информации в ответ на запрос.

Отправка пакетов с ответами подсистем

В ответ на запросы SRC подсистема должна отправлять структуру **srcrep**, описанную в файле **/usr/include/spc.h**. Ответ должен содержаться в структуре **svrreply**, вложенной в структуру **srcrep**:

```
struct svrreply
{
      short rtncode; /*код возврата подсистемы*/
      short objtype; /*SUBSYSTEM или  SUBSERVER*/
      char objtext[65]; /*описание объекта*/
      char objname[20]; /*имя объекта*/
      char rtnmsg[256]; /*возвращенное сообщение*/
};
```

Для отправки пакета с ответом предназначена функция **srcsrpy**.

Создание ответа

Для создания ответа подсистемы выполните следующую процедуру:

1. Укажите в поле `rtncode` код ошибки SRC. Для возврата собственного сообщения NLS укажите в поле `rtncode` значение **SRC_SUBMSG**.
2. Укажите в поле `objtype` константу SUBSYSTEM, если ответ отправляется от подсистемы, или кодовый знак субсервера, если ответ отправляется от субсервера.
3. Укажите в поле `objname` имя подсистемы, тип субсервера или объект субсервера, от которого отправляется ответ.
4. Укажите в поле `rtnmsg` свое сообщение NLS.
5. Укажите соответствующее значение в параметре *Continued* команды **srcsrpy**. Дополнительная информация приведена в разделе "Пакеты продолжения srcsrpy".

   Примечание: При отправке последнего пакета в параметре *Continued* функции **srcsrpy** должно быть указано значение END.

Пакеты с продолжением srcsrpy

Ответ подсистемы на запрос SRC представляет собой набор пакетов с продолжением. Существует два типа пакетов с продолжением: информационное сообщение и пакет с ответом.

Информационное сообщение не возвращается клиенту. Оно записывается в стандартный вывод клиента. Сообщение должно представлять собой текст NLS с маркерами сообщения, составленный подсистемой-отправителем. Для отправки такого пакета с продолжением укажите в параметре *Continued* функции **srcsrpy** значение CONTINUED.

Примечание: В ответ на запрос действия STOP не разрешается отправлять пакеты с продолжением. В ответ на все остальные запросы SRC, полученные подсистемой, может быть отправлено информационное сообщение.

Пакет с ответом возвращается клиенту для дальнейшей обработки. Формат этого пакета должен быть согласован между инициатором и подсистемой. Примером запроса, в ответ на который может быть отправлено такое продолжение, может служить запрос на получение информации о состоянии. При получении подсистемой этого запроса укажите в параметре *Continued* функции **srcsrpy** значение STATCONTINUED. После отправки всего отчета о состоянии или всех пакетов с ответом подсистемы укажите в параметре *Continued* функции **srcsrpy** значение END. Клиенту будет передан пакет, означающий завершение ответа.

Возврат подсистемами пакетов с ошибками SRC

Подсистемы должны отправлять пакеты с информацией обо всех ошибках SRC и других ошибках.

Информация об ошибке SRC должна возвращаться подсистемой в виде структуры **svrreply**, вложенной в структуру **srcrep**. В поле `objname` этой структуры должно быть указано имя подсистемы, тип субсервера или объект субсервера, в котором возникла ошибка. Если сообщение NLS, связанное с данной ошибкой SRC, не содержит маркеров, возвращается пакет с краткой информацией об ошибке. Это означает, что пакет содержит только номер ошибки SRC. Если с данной ошибкой связаны маркеры, то должно быть возвращено сообщение NLS из каталога сообщений.

При возникновении ошибки, не связанной с SRC, должен быть отправлен пакет со структурой **svrreply**, в которой полю `rtncode` присвоено значение SRC_SUBMSG, а в поле `rtnmsg` указано сообщение NLS подсистемы. Поле `rtnmsg` записывается в стандартный вывод клиента.

Ответ на запросы о трассировке

Подсистема не обязана поддерживать команды **traceson** и **tracesoff**. Однако поддержка этих команд позволяет выполнять трассировку подсистемы и субсервера.

Запросы на трассировку подсистемы отправляются в виде структуры **subreq**, полю `action` которой присвоено значение TRACE, а полю `object` - значение SUBSYSTEM. В параметре `parm1` указывается тип трассировки (LONGTRACE или SHORTTRACE), а в параметре `parm2` указывается, нужно ли включить или выключить трассировку (значение TRACEON или TRACEOFF).

Если подсистема получает пакет с запросом на трассировку, в котором `parm1` равен SHORTTRACE, а `parm2` равен TRACEON, то подсистема должна включить функцию трассировки. Если подсистема получит пакет с запросом на трассировку, в котором `parm1` равен LONGTRACE, а параметр `parm2` равен TRACEON, то подсистема должна включить функцию подробной трассировки. При получении пакета с запросом, в котором `parm2` будет равен TRACEOFF, подсистема должна выключить трассировку.

В запросе на трассировку субсервера поле `action` **subreq** равно TRACE, а поле `object` **subreq** содержит кодовый знак субсервера, для которого должна быть включена трассировка. В параметре `parm1` указывается тип трассировки (LONGTRACE или SHORTTRACE), а в параметре `parm2` указывается, нужно ли включить или выключить трассировку (значение TRACEON или TRACEOFF).

Если подсистема получает пакет с запросом на трассировку субсервера, в котором `parm1` равен SHORTTRACE, а `parm2` равен TRACEON, то подсистема должна включить трассировку субсервера. Если подсистема получает пакет с запросом на трассировку субсервера, в котором `parm1` равен LONGTRACE, а `parm2` равен TRACEON, то она должна включить подробную трассировку субсервера. При получения пакета с запросом на трассировку субсервера, в котором `parm2`равен TRACEOFF, подсистема должна выключить трассировку субсервера.

Ответ на запросы об обновлении

Не все подсистемы поддерживают запросы на обновление. Подсистемы, поддерживающие команду **refresh**, должны отвечать на нее SRC следующим образом:

- В запросе на обновление подсистемы поле `action` структуры **subreq** равно REFRESH, а поле `object` равно SUBSYSTEM. При обновлении подсистемы не применяются параметры `parm1` и `parm2`.
- При получении запроса на обновление подсистема должна изменить свою конфигурацию.

Определение подсистемы в SRC

Подсистемы определяются в классе объектов SRC в виде объектов подсистем. Субсерверы определяются в базе данных конфигурации SRC в виде объектов типа субсервер.

Структуры, связанные с каждым типом объектов, определены в файле **sys/srcobj.h**.

Для создания объекта подсистемы предназначена команда **mkssys** и процедура **addssys**. Объект типа субсервер создается с помощью команды **mkserver**. При выполнении команд или процедур настройки не обязательно указывать все опции и параметры. Для некоторых из них SRC предлагает значения по умолчанию. Вам нужно заполнить только обязательные поля, и, при необходимости, изменить некоторые значения по умолчанию.

Для добавления и изменения дескрипторов из командной строки можно создать сценарий оболочки. Кроме того, это можно сделать с помощью интерфейса C. В нем предусмотрены команды и функции настройки и изменения объектов SRC.

Примечание: Вариант программного интерфейса приведен только для удобства.

Предусмотрены следующие команды:

Команда	Описание
mkssys	Добавляет определение подсистемы в базу данных конфигурации SRC.
mkserver	Добавляет определение субсервера в базу данных конфигурации SRC.
chssys	Изменяет определение подсистемы в базе данных конфигурации SRC.
chserver	Изменяет определение субсервера в базе данных конфигурации SRC.
rmssys	Удаляет определение подсистемы из базы данных конфигурации SRC.
rmserver	Удаляет определение субсервера из базы данных конфигурации SRC.

В интерфейсе C предусмотрены следующие функции:

Функция	Описание
addssys	Добавляет определение подсистемы в базу данных конфигурации SRC
chssys	Изменяет определение подсистемы в базе данных конфигурации SRC
defssys	Инициализирует новое определение подсистемы значениями по умолчанию
delssys	Удаляет определение подсистемы из базы данных конфигурации SRC **Примечание:** Объектный код функции **chssys** должен быть запущен в системе из той же группы.
getssys	Получает определение подсистемы из базы данных конфигурации SRC
getsubsvr	Получает определение субсервера из базы данных конфигурации SRC

Команды **mkssys** и **mkserver** перед добавлением и изменением значений в командной строке вызывают функцию **defssys** для определения значений по умолчанию подсистемы или субсервера.

Для получения данных из файлов конфигурации SRC главная программа SRC и подсистема применяют функции **getssys** и **getsubsvr**.

Список дополнительных функций SRC

Для поддержки обмена данными между SRC и подсистемами предназначены следующие функции:

Функция	Описание
src_err_msg	Возвращает текст сообщения об ошибке SRC, обнаруженной библиотечной процедурой SRC. (См. также процедуру **src_err_msg_r** с защитой нитей)
srcsbuf	Запрашивает информацию о состоянии подсистемы в формате для печати. (См. также функцию **srcsbuf_r** с защитой нитей)
srcsrqt	Отправляет сообщение или запрос подсистеме. (См. также функцию **srcsrqt_r** с защитой нитей)
srcstat	Запрашивает краткую информацию о состоянии подсистемы. (См. также функцию **srcstat_r** с защитой нитей)
srcstathdr	Получает заголовок отчета о состоянии SRC.
srcstattxt	Получает текстовое описание кода состояния SRC. (См. также функцию **srcstattxt_r** с защитой нитей)
srcstop	Отправляет запрос на завершение работы подсистемы.
srcstrt	Отправляет запрос на запуск подсистемы.

Функция трассировки

Трассировщик служит для локализации неполадок в системе путем отслеживания указанных системных событий или процессов. Можно отслеживать следующие события: вызов и возврат из выбранных функций, функций ядра, функций расширений ядра и обработчиков прерываний.

Предусмотрена возможность трассировки набора работающих процессов или нитей, а также запуска произвольной программы в режиме трассировки.

Трассировщик ведет системный файл протокола трассировки. Трассировщик содержит команды для включения трассировки, управления трассировкой и создания отчетов о трассировке. Приложения и расширения ядра могут заносить в протокол дополнительные события с помощью ряда специальных функций.

Дополнительная информация о трассировщике приведена в следующих разделах:

Основные сведения о планировщике

Трассировщик находится в наборе файлов **bos.sysmgt.trace**. Для того чтобы узнать, установлен ли данный набор файлов, введите следующую команду:

```
lslpp -l | grep bos.sysmgt.trace
```

Если в данные вывода содержат строку **bos.sysmgt.trace** , то набор файлов установлен, в противном случае, его необходимо установить.

Трассировщик системы сохраняет информацию о событиях трассировки, которую затем можно отформатировать командой trace report. События трассировки включены в ядро или код приложения, но отслеживаются только во время трассировки.

Запустить трассировку позволяет команда **trace** и функция **trcstart**. Для завершения трассировки предназначены команда **trcstop** и функция **trcstop**. Для приостановки и возобновления трассировки предназначены команды **trcoff** и **trcon**, а также функции **trcoff** и **trcon**.

После того как трассировка была остановлена с помощью команды **trcstop**, команда **trcrpt** позволяет создать отчет. Формат записей, применяемый данной командой, задается в файле шаблона **/etc/trcfmt**. Для установки шаблонов предназначена команда **trcupdate**. Дополнительная информация о шаблонах приведена в описании команды **trcupdate**.

Управление трассировкой

Команда **trace** позволяет запустить трассировку событий системы и задать размер буфера и файла протокола трассировки. Информация об этой команде приведена в разделе, посвященном демону **trace**, книги Command's Reference.

Существует три способа сбора данных трассировки.

1. По умолчанию данные непрерывно собираются в два буфера, причем пока данные трассировки заносятся в один буфер, данные другого буфера записываются в файл протокола. Если файл протокола достигает максимального размера, новые данные записываются поверх наиболее ранних записей.
2. При циклическом способе сбор данных выполняется непрерывно, однако файл протокола записывается только после завершения трассировки. Этот способ применяется при отладке локализованной ошибки, если необходимо получить данные в момент ее возникновения. Трассировку можно запустить в любой момент, а затем завершить ее сразу после возникновения ошибки, таким образом будут получены данные о событиях, предшествовавших возникновению ошибки. Для применения этого способа трассировки необходимо указать флаг демона трассировки **-l**.
3. При третьем способе применяется один буфер, трассировка завершается при его заполнении, после чего данные из буфера записываются в файл протокола. В этот момент трассировка не завершается, а отключается (как если бы была введена команда **trcoff**). Обычно на этом этапе трассировка завершается командой **trcstop**. Этот способ применяется, если до сбора данных трассировка не должна заменять ввод-вывод и содержимое буферов. Для применения этого способа трассировки необходимо указать флаг **-f**.

Обычно команда trace запускается в асинхронном режиме, т.е. после ее запуска можно продолжить работу обычным образом. Для запуска трассировки в асинхронном режиме нужно указать флаг **-a** или **-x**. Если будет указан флаг **-a**, для прекращения трассировки нужно будет выполнить команду **trcstop**. Если будет указан флаг **-x**, трассировка будет автоматически прекращена после завершения работы программы.

Рекомендуется ограничивать отслеживаемую информацию. Флаги **-j события** и **-k события** позволяют указать наборы событий, которые будут включены (**-j**) в трассировку или исключены (**-k**) из нее.

Примечание: Ограничивая трассировку отдельными процессами или нитями, вы ограничиваете объем информации, доступной для трассировки.

Для просмотра имен программ, связанных с точками трассировки, необходимо включить соответствующие точки трассировки. Их можно указать с помощью группы событий трассировки **tidhk**. Например, для трассировки точки **mbuf** 254 и просмотра имен программ необходимо ввести следующую команду **trace**:

```
trace -aJ tidhk -j 254
```

Выполняется трассировка. Для завершения трассировки введите следующую команду:

```
trcstop
trcrpt -O exec=on
```

Опция trcrpt **-O exec=on** позволяет просмотреть имена программ. За дополнительной информацией обратитесь к описанию команды **trcrpt**.

В некоторых случаях рекомендуется указывать размер буфера и максимальный размер файла протокола. Для буферов трассировки необходима физическая память, так как запись точек трассировки должна выполняться без подкачки. После достижения максимального размера файла протокола данные трассировки записываются поверх наиболее ранних записей. Флаги **-T размер** и **-L размер** позволяют указать в байтах размер буферов и максимальный размер данных трассировки в файле протокола.

Примечание: Так как трассировщик резервирует для буферов сбора данных определенный объем оперативной памяти системы, трассировка может оказать негативное влияние на производительность системы с небольшим объемом оперативной памяти. Если трассируемому приложению достаточно оперативной памяти, или объем памяти, используемой функцией трассировки, составляет небольшой процент от общего объема оперативной памяти системы, влияние трассировки на производительность будет небольшим. Если значение не указано, применяется размер, заданный по умолчанию.

Управлять трассировкой можно и из приложения. Дополнительная информация приведена в описаниях команд **trcstart** и **trcstop**.

Запись данных событий трассировки

Существует два типа данных трассировки.

Общие данные
: Состоят из слова данных, буфера скрытых данных и длины этих данных. Этот тип применяется для трассировки таких элементов, как полные имена файлов. См. пункт Каналы трассировки общего назначения в разделе **Общие сведения о трассировщике**. Он расположен в главе **Трассировщик**.

  Примечание: Трассировка отдельных процессов и нитей поддерживается только для канала 0. Она не поддерживается для общих каналов трассировки.

Специальные данные
: Это стандартные данные трассировки операционной системы AIX. Записи этого типа состоят из слова точки трассировки и до пяти слов данных трассировки. В 64-разрядных приложениях это 8-байтовые слова. При программировании на языке C для записи специальных данных предназначены макрокоманды TRCHKL0 - TRCHKL5 и TRCHKL0T - TRCHKL5T, определенные в файле **/usr/include/sys/trcmacros.h**. Если по каким-то причинам эти макрокоманды применять нельзя, ознакомьтесь с описанием функции **utrchook**.

Создание отчета трассировки

Полное описание команды **trcrpt** приведено в разделе **Команда trcrpt**. Эта команда позволяет создать отчет для пользователя на основе файла протокола, сгенерированного командой **trace**. По умолчанию команда форматирует данные из файла протокола **/var/adm/ras/trcfile**. Выходные данные команды **trcrpt** отправляется в стандартный вывод.

Для создания отчета трассировки из файла по умолчанию и его записи в /tmp/rptout введите команду

trcrpt >/tmp/rptout

Для создания отчета трассировки с названиями программ и именами системных вызовов из файла /tmp/tlog и его записи в файл /tmp/rptout введите команду

trcrpt -O exec=on,svc=on /tmp/tlog >/tmp/rptout

Получение данных трассировки из дампа

Если во время трассировки был создан системный дамп, данные трассировки можно получить с помощью команды **trcdead**. Для того чтобы сохранить в системе файл протокола трассировки по умолчанию, укажите опцию **-o** ***файл-вывода***.

Пример:

```
trcdead -o /tmp/tlog /var/adm/ras/vmcore.0
```

создает файл протокола трассировки /tmp/tlog, который затем можно отформатировать командой

```
trcrpt /tmp/tlog
```

Команды трассировщика

Перечисленные команды являются частью трассировщика:

Команда	Функция
trace	Запускает трассировку системных событий. С помощью этой команды можно определять размер и параметры файла протокола трассировки, а также внутренних буферов трассировки, в которых хранятся данные об отслеживаемых событиях.
trcdead	Извлекает трассировочную информацию из системного дампа. Если в момент останова системы работали средства трассировки, то содержимое внутренних буферов трассировки будет включено в дамп. Эта команда извлекает данные об отслеживаемых событиях из файла дампа и записывает их в файл протокола трассировки.
trcnm	Генерирует список имен ядра для команды **trcrpt**. Этот список формируется из таблицы имен и таблицы имен загрузчика для объектного файла. С помощью файла имен ядра команда **trcrpt** выполняет преобразование адресов при создании отчета на основе файла протокола трассировки. **Примечание:** Вместо опции **trcnm** рекомендуется указывать опцию **-n trace**. Тогда список имен будет сохранен в файле протокола трассировки, а не в отдельном файле, и будет содержать имена из расширений ядра.
trcrpt	Формирует отчеты об отслеживаемых событиях, занесенных в файл протокола трассировки. При вызове этой команды можно указать, какие события следует включить в отчет или, наоборот, не включать в него, а также определить формат вывода. Способ интерпретации командой **trcrpt** данных, относящихся к каждому событию, определяется шаблонами, хранящимися в файле **/etc/trcfmt**.
trcstop	Прекращает трассировку системных событий.
trcupdate	Обновляет шаблоны форматирования трассировки в файле **/etc/trcfmt**. При добавлении новых приложений или расширений ядра, регистрирующих события трассировки, в файл **/etc/trcfmt** необходимо добавить шаблоны для этих событий. Шаблоны форматирования трассировки определяют способ интерпретации данных, относящихся к каждому событию, в команде **trcrpt**. Программные продукты, выполняющие регистрацию событий, обычно запускают команду **trcupdate** во время установки.

Вызовы и функции трассировщика

Трассировщик включает в себя следующие вызовы и функции:

Функция	Описание
trcgen, **trcgent**	Регистрирует наступление отслеживаемых событий в виде записей, состоящих более чем из пяти слов данных. С помощью функции **trcgen** можно записать информацию о событии в канал трассировки системных событий (канал 0) или в канал трассировки общего назначения (каналы 1-7). Номер канала передается в качестве параметра функции при записи информации о событии. Функция **trcgent** добавляет к данным о событии текущее системное время. При работе с AIX 5L версии 5.3 с технологическим уровнем обслуживания 5300-05 и выше системное время всегда добавляется к данным события независимо от применяемой функции. При работе с ядром следует применять функции **trcgenk** и **trcgenkt**. При программировании на языке C всегда следует применять макрокоманды **TRCGEN** и **TRCGENK**.
utrchook, **utrchook64**	Регистрирует наступление отслеживаемых событий в виде записей, состоящих не более чем из пяти слов данных. Эти функции позволяют записать информацию о событии в канал трассировки системных событий (канал 0). При программировании ядра можно использовать функции **trchook** и **trchook64**. При программировании на языке C всегда следует применять макрокоманды **TRCHKL0** - **TRCHKL5** и **TRCHKL0T** - **TRCHKL5T**. Если эти макрокоманды не используются, то необходимо создать собственное слово с данными о событии трассировки. Соответствующий формат описан в файле **/etc/trcfmt**. Учтите, что слова с информацией о событии 32-разрядной и 64-разрядной трассировки отличаются.
trcoff	Приостанавливает регистрацию отслеживаемых событий в канале трассировки системных событий (канал 0) или в канале трассировки общего назначения (каналы 1-7). При этом канал трассировки остается активным. Возобновить регистрацию событий можно с помощью функции **trcon**.
trcon	Запускает регистрацию отслеживаемых событий в канале трассировки системных событий (канал 0) или в канале трассировки общего назначения (каналы 1-7). Однако канал трассировки необходимо предварительно активизировать с помощью команды **trace** или функции **trcstart**. Приостановить регистрацию событий можно с помощью команды **trcoff**.
trcstart	Библиотечный интерфейс команды trace. Возвращает номер канала запускаемой трассировки. Трассировка общего назначения может быть запущена в каналах 1,2,3,4,5,6 и 7. В иных случаях номер канала равен 0.
trcstop	Освобождает и дезактивирует канал трассировки общего назначения.

Файлы трассировщика

Файл	Описание
/etc/trcfmt	Содержит шаблоны форматирования трассировки, с помощью которых команда **trcrpt** определяет способ интерпретации данных, относящихся к каждому событию.
/var/adm/ras/trcfile	Это файл протокола трассировки по умолчанию. В команде **trace** можно указать другое имя файла протокола трассировки.
/usr/include/sys/trchkid.h	Содержит определения идентификаторов точек трассировки.
/usr/include/sys/trcmacros.h	Содержит набор часто применяемых макроопределений для регистрации отслеживаемых событий.

Записи о событиях трассировки

Формат данных о событиях трассировки описан в файле **/etc/trcfmt**.

Идентификаторы точек трассировки

Идентификатор точки трассировки представляет собой трех- или четырехзначное шестнадцатеричное число, соответствующее отслеживаемому событию. В версиях, более ранних, чем AIX 7.1, а также в 32-разрядных приложениях под AIX 7.1 и выше, могут применяться только трехзначные идентификаторы точки трассировки. При использовании макросов трассировки, таких как **TRCHKL1**, точка трассировки определяется следующим образом:

```
hhh00000
```

где hhh - идентификатор точки трассировки.

В 64-разрядных приложениях и процедурах ядра под AIX 7.1 и выше можно использовать трех- и четырехзначные идентификаторы. При использовании макросов трассировки, таких как **TRCHKL1**, точка трассировки определяется следующим образом:

```
hhhh0000
```

где hhhh - идентификатор точки трассировки.

Примечание: Если используется четырехзначный идентификатор, и он меньше, чем 0x1000, то наименьшая по значимости цифра должна быть 0 (в форме 0x0hh0).

Трехзначный идентификатор имеет 0 в качестве наименьшей цифры. Таким образом, 32-разрядный идентификатор точки трассировки hhh равен 64-разрядному идентификатору hhh0.

Большинство идентификаторов точек трассировки определены в файле **/usr/include/sys/trchkid.h**. В пользовательских 64-разрядных приложениях могут применяться значения в диапазоне 0x010-0x0FF. В пользовательских 32-разрядных приложениях могут применяться значения в диапазоне 0x010-0x0FF. Все остальные значения зарезервированы для системных целей. Список определенных в данный момент идентификаторов точек трассировки можно просмотреть с помощью команды **trcrpt -j**.

Каналы трассировки общего назначения

Трассировщик может поддерживать до восьми активных сеансов трассировки одновременно. Каждый сеанс трассировки работает с отдельным каналом специального файла мультиплексной трассировки **/dev/systrace**. Канал 0 служит для регистрации системных событий. Трассировка системных событий запускается командой **trace** и завершается командой **trcstop**. При трассировке отдельных процессов или нитей, а также при трассировке программ используется только канал 0. Каналы 1-7 называются каналами трассировки общего назначения и могут использоваться подсистемами для трассировки других событий, например, событий передачи данных.

Для инициализации канала трассировки общего назначения подсистема вызывает функцию **trcstart**, которая активизирует канал трассировки и возвращает его номер. После этого модули подсистемы могут регистрировать события с помощью макрокоманд **TRCGEN** и **TRCGENT**, либо, при необходимости, с помощью функций **trcgen**, **trcgent**,**trcgenk** и **trcgenkt**. Номер канала, возвращаемый функцией **trcstart**, должен передаваться в эти функции в числе прочих параметров. Подсистема может приостановить регистрацию с помощью функции **trcoff**, возобновить ее с помощью функции **trcon** и деактивировать канал трассировки с помощью функции **trcstop**. Для активизации и деактивизации трассировки в подсистеме должен быть предусмотрен пользовательский интерфейс.

Каналы трассировки работают с общими идентификаторами точек трассировки, большая часть которых хранится в файле **/usr/include/sys/trchkid.h**, и шаблонами форматирования трассировки, хранящимися в файле **/etc/trcfmt**.

Информация, связанная с данной:

trace

trcdead

trcnm

trcrpt

trcstop

trcupdate

Запуск трассировщика

Выполните следующие действия для настройки и запуска системной трассировки:

Настройка команды trace

Команда **trace** запускает трассировку системных событий, а также управляет файлом протокола трассировки и буферами трассировки, в которых хранятся данные об отслеживаемых событиях. Информация о синтаксисе этой команды приведена в разделе демон трассировки.

Запись данных событий трассировки

Запись о наступлении каждого отслеживаемого события имеет следующий формат: слово, содержащее идентификатор точки трассировки и ее тип, затем переменное количество слов данных трассировки и, наконец, необязательное системное время. Слово, содержащее идентификатор и тип точки трассировки, называется ключевым словом. Оставшиеся два байта ключевого слова называются данными точки трассировки и могут применяться для регистрации события.

Идентификаторы точек трассировки

Идентификатор точки трассировки представляет собой трех- или четырехзначное шестнадцатеричное число, соответствующее отслеживаемому событию. В версиях, более ранних, чем AIX 6.1, а также в 32-разрядных приложениях под AIX 6.1 и выше, могут применяться только трехзначные идентификаторы точки трассировки. При использовании макросов трассировки, таких как **TRCHKL1**, точка трассировки определяется следующим образом:

```
hhh0000
```

где hhh - идентификатор точки трассировки.

В 64-разрядных приложениях и процедурах ядра под AIX 6.1 и выше можно использовать трех- и четырехзначные идентификаторы точек трассировки. При использовании макросов трассировки, таких как **TRCHKL1**, точка трассировки определяется следующим образом:

```
hhhh0000
```

где hhhh - идентификатор точки трассировки.

Примечание: Если используется четырехзначный идентификатор, и он меньше, чем 0x1000, то наименьшая по значимости цифра должна быть 0 (в форме 0x0hh0).

Трехзначный идентификатор имеет 0 в качестве наименьшей цифры. Таким образом, 32-разрядный идентификатор точки трассировки hhh равен 64-разрядному идентификатору hhh0.

Большинство идентификаторов точек трассировки определены в файле **/usr/include/sys/trchkid.h**. В 64-разрядных приложениях могут применяться значения в диапазоне 0x010-0x0FF. В пользовательских 32-разрядных приложениях могут применяться значения в диапазоне 0x010-0x0FF. Все остальные значения зарезервированы для системных целей. Список определенных в данный момент идентификаторов точек трассировки можно просмотреть с помощью команды **trcrpt -j**.

Применение каналов трассировки общего назначения

Трассировщик может поддерживать до восьми активных сеансов трассировки одновременно. Каждый сеанс трассировки работает с отдельным каналом специального файла мультиплексной трассировки **/dev/systrace**. Канал 0 служит для регистрации системных событий. Трассировка системных событий запускается командой **trace** и завершается командой **trcstop**. Каналы 1-7 называются каналами трассировки общего назначения и могут использоваться подсистемами для трассировки других событий, например, событий передачи данных.

Для инициализации канала трассировки общего назначения подсистема вызывает функцию **trcstart**, которая активизирует канал трассировки и возвращает его номер. После этого модули подсистемы могут регистрировать события с помощью макросов **TRCGEN** или **TRCGENT**, либо функций **trcgen**, **trcgent**, **trcgenk** или **trcgenkt**. Номер канала, возвращаемый функцией **trcstart**, должен передаваться в эти функции в числе прочих параметров. Подсистема может приостановить регистрацию с помощью функции **trcoff**, возобновить ее с помощью функции **trcon** и деактивировать канал трассировки с помощью функции **trcstop**. События, отслеживаемые в каждом канале, должны заноситься в отдельный файл протокола **трассировки**, который по умолчанию называется **/var/adm/ras/trcfile.*n***, где *n* - это номер катала. Для активизации и деактивизации трассировки в подсистеме должен быть предусмотрен пользовательский интерфейс.

Запуск трассировки

Ниже описаны способы запуска трассировки.

- Запустите трассировку с помощью команды **trace**.

  Запустите трассировку в асинхронном режиме. Например:

  ```
  trace -a
  mycmd
  trcstop
  ```

  В асинхронном режиме трассировка выбранных системных событий (например, команды **mycmd**) запускается с помощью демона **trace**. Завершить трассировку позволяет команда **trcstop**.

  ИЛИ

  Запустите трассировку в интерактивном режиме. Например:

  ```
  trace
  ->!mycmd
  ->quit
  ```

  При работе с трассировкой в интерактивном режиме необходимо перейти к командной строке **->** и с помощью подкоманд **трассировки** (таких как **!**) выполнять трассировку выбранных системных событий. Завершить трассировку можно с помощью подкоманды **quit**.

- Вызовите команду **smit trace** и выберите опцию **Запустить трассировку**.

  ```
  smit trace
  ```

Завершение трассировки

Ниже описаны способы завершения трассировки.

- Если **трассировка** выполняется в асинхронном режиме, введите в командной строке команду **trcstop**:

  ```
  trace -a
  mycmd
  trcstop
  ```

  В асинхронном режиме трассировка выбранных системных событий (например, команды **mycmd**) запускается с помощью демона **trace**. Завершить трассировку позволяет команда **trcstop**.

- Если команда **trace** была запущена в интерактивном режиме из командной строки, то введите подкоманду **quit**:

  ```
  trace
  ->!mycmd
  ->quit
  ```

Об интерактивном режиме свидетельствует командная строка **->**. Подкоманды **trace** (например, **!**) позволяют выполнять трассировку выбранных системных событий. Завершить трассировку можно с помощью подкоманды **quit**.

- Вызовите команду **smit trace** и выберите опцию **Завершить трассировку**.

  ```
  smit trace
  ```

Создание отчета трассировки

Ниже описаны различные способы создания отчета о трассировке событий.

- С помощью команды **trcrpt**:

  ```
  trcrpt>/tmp/NewFile
  ```

  В предыдущем примере данные файла протокола трассировки форматируются и записываются в файл **/tmp/newfile**. Команда **trcrpt** получает данные из файла протокола трассировки, форматирует записи трассировки и создает отчет.
- С помощью команды **smit trcrpt**:

  ```
  smit trcrpt
  ```

Пользовательское приложение трассировки

В этом разделе описана трассировка пользовательских приложений.

Операция трассировки зависит от трех различных по логике процессов: отслеживаемый процесс, процесс контроллера и процесс анализатора. Один процесс может одновременно быть отслеживаемым процессом, процессом контроллера и процессом анализатора. Когда запущен отслеживаемый процесс, то при достижении точки трассировки событие трассировки записывается в потоки трассировки, созданные для этого процесса, если идентификатор типа событий трассировки, связанный с этим процессом, не исключен фильтром.

Процесс контроллера управляет записью событий трассировки в потоки трассировки. Для активного потока трассировки процесс контроллера выполняет следующие операции:

- Инициализация атрибутов потока трассировки.
- Создание потока трассировки для указанного отслеживаемого процесса с помощью инициализированных атрибутов.
- Запуск и остановка трассировки для потока трассировки.
- Фильтр типа событий трассировки, подлежащего записи.
- Завершение потока трассировки.

Процесс анализатора получает отслеживаемые события либо во время выполнения, когда поток трассировки активен и выполняет запись событий трассировки, либо после открытия протокола трассировки, ранее созданного и закрытого.

Функции **posix_trace_create**, **posix_trace_create_withlog** и **posix_trace_open** создают идентификатор потока трассировки. Функции **posix_trace_create** и **posix_trace_create_withlog** используются только процессом контроллера. Функция **posix_trace_open** используется только процессом анализатора.

Отслеживаемый процесс содержит в себе преобразование имен событий трассировки в идентификаторы типов событий трассировки, определенные для процесса. Активное событие трассировки записывает стандартные, определенные системой типы событий трассировки, например, POSIX_TRACE_START, и типы событий трассировки, определенные для отслеживаемых процессов, но не исключенные фильтром потока трассировки. Для того, чтобы определить преобразование, нужно вызвать функцию **posix_trace_eventid_open** из инструментального приложения или вызвать функцию

posix_trace_trid_eventid_open из процесса контроллера. Для предварительно записанного потока трассировки, список типов событий трассировки поступает из предварительно записанного протокола трассировки.

Функции трассировки можно применять при отладке предварительно реализованного приложения и для аварийного анализа сбоев. Для отладки предварительно реализованного кода могут потребоваться функции предварительной фильтрации, которые помогут избежать переполнения потока трассировки и обеспечат четкую направленность на получение ожидаемой информации. Для проведения аварийного анализа сбоев требуются расширенные функции трассировки, позволяющие записывать информацию любого типа.

События, подлежащие трассировке, можно разбить на два класса:

- Пользовательские события трассировки, сгенерированные реализованным приложением.
- Системные события трассировки, сгенерированные операционной системой в соответствии с операцией управления трассировкой.

В файле, связанном с активным потоком трассировки, поля `st_ctime` и `st_mtime` помечены как подлежащие обновлению при каждом изменении этого файла операциями трассировки.

В файле, связанном с потоком трассировки, поле `st_atime` помечено как подлежащее обновлению при каждом считывании информации из этого файла, инициированном любой из операций трассировки.

Если приложением выполняется какая-либо операция над дескриптором файла, связанным с активным или предварительно записанным потоком трассировки, результаты не будут определены до тех пор, пока не будет вызвана функция **posix_trace_shutdown** или **posix_trace_close** для данного потока трассировки.

Структуры данных трассировки

В этом разделе рассмотрена структура данных трассировки.

Файл заголовка <trace.h> определяет структуры **posix_trace_status_info** и **posix_trace_event_info**.

Структура posix_trace_status_info

Для оптимизации управления потоком трассировки вызовите функцию **posix_trace_get_status**, позволяющую в динамическом режиме получать информацию о текущем состоянии активного потока трассировки.

Структура **posix_trace_status_info**, определенная в файле <trace.h>, состоит из следующих элементов:

Тип элемента	Имя элемента	Описание
int	posix_stream_status	Рабочий режим потока трассировки.
int	posix_stream_full_status	Полное состояние потока трассировки.
int	posix_stream_overrun_status	Указывает на то, не были ли в потоке трассировки утеряны события трассировки.
int	posix_stream_flush_status	Указывает на то, выполняется ли в данный момент очистка.
int	posix_stream_flush_error	Указывает на то, возникали ли ошибки во время последней операции очистки.
int	posix_log_overrun_status	Указывает на то, не были ли в протоколе трассировки утеряны события трассировки.
int	posix_log_full_status	Полное состояние протокола трассировки.

Элемент **posix_stream_status** указывает на рабочий режим потока трассировки. Значение этого элемента может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_RUNNING
Выполняется трассировка. Поток трассировки принимает события трассировки.

POSIX_TRACE_SUSPENDED
Поток трассировки не принимает события трассировки. Операция трассировки не начата или остановлена, либо вследствие вызова функции **posix_trace_stop**, либо потому, что ресурсы трассировки исчерпаны.

Элемент **posix_stream_full_status** указывает на состояние заполненности потока трассировки. Значение этого элемента может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_FULL
Пространство, выделенное в потоке трассировки для событий трассировки, исчерпано.

POSIX_TRACE_NOT_FULL
Пространство в потоке трассировки не заполнено.

Сочетание элементов **posix_stream_status** и **posix_stream_full_status** указывает на фактическое состояние потока. Это состояние можно интерпретировать следующим образом:

POSIX_TRACE_RUNNING и POSIX_TRACE_NOT_FULL
Трассировка выполняется, и есть свободное пространство для записи последующих событий трассировки.

POSIX_TRACE_RUNNING и POSIX_TRACE_FULL
Трассировка выполняется, и поток трассировки заполнен событиями трассировки. Такое состояние может возникнуть, если для стратегии заполненного потока задано значение POSIX_TRACE_LOOP. Поток трассировки содержит в себе события трассировки, записанные во время перемещения окна времени предыдущих событий трассировки, и некоторые события предшествующей трассировки могут быть перезаписаны и утрачены.

POSIX_TRACE_SUSPENDED и POSIX_TRACE_NOT_FULL
Трассировка не начата, трассировка остановлена функцией **posix_trace_stop**, либо с помощью функции **posix_trace_clear** выполнена очистка трассировки.

POSIX_TRACE_SUSPENDED и POSIX_TRACE_FULL
Трассировка останавливается. Причина заключается в том, что для атрибута стратегии заполненного потока задано значение POSIX_TRACE_UNTIL_FULL, и ресурсы трассировки исчерпаны, либо в вызове функции **posix_trace_stop** при исчерпанных ресурсах трассировки.

Элемент **posix_stream_overrun_status** указывает на то, не были ли в потоке трассировки утеряны события трассировки. Значение этого элемента может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_OVERRUN
По крайней мере одно событие трассировки утрачено и не записано в поток трассировки.

POSIX_TRACE_NO_OVERRUN
Не утеряно ни одного события трассировки.

При создании соответствующего потока трассировки, элемент **posix_stream_overrun_status** получает значение POSIX_TRACE_NO_OVERRUN. Когда происходит переполнение, устанавливается состояние POSIX_TRACE_OVERRUN.

Переполнение происходит в следующих ситуациях:

- Задана стратегия POSIX_TRACE_LOOP, и записанное событие трассировки перезаписано.
- Задана стратегия POSIX_TRACE_UNTIL_FULL, и поток трассировки на момент генерирования события трассировки заполнен.
- Задана стратегия POSIX_TRACE_FLUSH, и во время очистки потока трассировки в протокол трассировки утеряно несколько событий трассировки.

После считывания значения элемента **posix_stream_overrun_status** оно сбрасывается на нуль.

Элемент **posix_stream_flush_status** указывает на то, выполняется ли в данный момент операция очистки. Значение этого элемента может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_FLUSHING
Выполняется очистка потока трассировки в протокол трассировки.

POSIX_TRACE_NOT_FLUSHING
Если в данный момент операция очистки не выполняется, для элемента **posix_stream_flush_status** устанавливается значение **POSIX_TRACE_NOT_FLUSHING**.

Элемент **posix_stream_flush_status** имеет значение **POSIX_TRACE_FLUSHING** в следующих случаях:

- Выполняется операция очистки вследствие вызова функции **posix_trace_flush**.
- Выполняется операция очистки, поскольку поток трассировки заполнен, а для атрибута стратегии заполненного потока задано значение POSIX_TRACE_FLUSH.

Если при очистке не возникает ошибок, элемент **posix_stream_flush_error** получает значение, равное нулю. Если в предыдущей операции очистки возникла ошибка, элемент **posix_stream_flush_error** получает значение, соответствующее первой возникшей ошибке. Если при операции очистки возникает несколько ошибок, используется значение первой ошибки, а остальные отбрасываются. После считывания значения элемента **posix_stream_flush_error** оно сбрасывается на нуль.

Элемент **posix_log_overrun_status** указывает на то, не были ли в протоколе трассировки утеряны события трассировки. Значение этого элемента может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_OVERRUN
Утеряно как минимум одно событие трассировки.

POSIX_TRACE_NO_OVERRUN
Не утеряно ни одного события трассировки.

При переполнении элемент **posix_log_overrun_status** получает значение POSIX_TRACE_OVERRUN. При создании соответствующего потока трассировки, элемент **posix_log_overrun_status** получает значение POSIX_TRACE_NO_OVERRUN.

После считывания значения элемента **posix_log_overrun_status** оно сбрасывается на нуль.

При создании активного потока трассировки, не имеющего протокола, с помощью функции **posix_trace_create**, элемент **posix_log_overrun_status** получает значение POSIX_TRACE_NO_OVERRUN.

Элемент **posix_log_full_status** указывает на состояние заполнения протокола трассировки, и может иметь одно из следующих значений, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_FULL
Пространство в протоколе трассировки заполнено.

POSIX_TRACE_NOT_FULL
Пространство в протоколе трассировки не заполнено.

Элемент **posix_log_full_status** является значимым, только если атрибут стратегии заполненного протокола имеет значение POSIX_TRACE_UNTIL_FULL или POSIX_TRACE_LOOP.

При создании активного потока трассировки, не имеющего протокола, с помощью функции **posix_trace_create**, элемент **posix_log_full_status** получает значение POSIX_TRACE_NOT_FULL.

Структура posix_trace_event_info

В структуре события трассировки **posix_trace_event_info** содержится информация о записанном событии трассировки. Функции **posix_trace_getnext_event**, **posix_trace_timedgetnext_event** и **posix_trace_trygetnext_event** возвращают эту функцию.

Структура **posix_trace_event_info**, определенная в файле заголовка <trace.h>, включает в себя следующие элементы:

Тип элемента	Имя элемента	Описание
trace_event_id_t	posix_event_id	Идентификация типа события трассировки.
pid_t	posix_pid	ИД процесса, генерирующего событие трассировки.
void*	posix_prog_address	Адрес вызова точки трассировки.
int	posix_truncation_status	Статус усечения данных, связанных с этим событием трассировки.
struct timespec	posix_timestamp	Время генерирования события трассировки.
pthread_t	posix_thread_id	ИД нити, генерирующей событие трассировки.

Элемент **posix_event_id** представляет идентификацию типа события трассировки. Непосредственно определить значение этого элемента нельзя. Для получения идентификации необходимо запустить одну из следующих функций:

- posix_trace_trid_eventid_open
- posix_trace_eventtypelist_getnext_id
- posix_trace_eventid_open

Для того, чтобы получить имя типа события трассировки, запустите функцию **posix_trace_eventid_get_name**.

posix_pid - это идентификатор отслеживаемого процесса, генерирующего событие трассировки. Если элемент **posix_event_id** является одним из системных событий трассировки, и данное событие трассировки не связано ни с одним из процессов, для элемента **posix_pid** устанавливается значение, равное нулю.

Для пользовательского события трассировки, элемент **posix_prog_address** представляет собой преобразованный адрес процесса и точку, в которой выполняется связанный вызов функции **posix_trace_event**.

Элемент **posix_truncation_status** определяет статус усечения данных, связанных с событием трассировки. Значение элемента **posix_truncation_status** может быть одним из следующих, определенных константами манифеста в заголовке <trace.h>:

POSIX_TRACE_NOT_TRUNCATED
Доступны все отслеживаемые данные.

POSIX_TRACE_TRUNCATED_RECORD
При генерировании события трассировки данные усекаются.

POSIX_TRACE_TRUNCATED_READ
Данные усекаются при считывании события трассировки из потока трассировки или протокола трассировки. Этим состоянием усечения переопределяется состояние POSIX_TRACE_TRUNCATED_RECORD.

Элемент **posix_timestamp** определяет время генерирования события трассировки. Используемые часы - CLOCK_REALTIME. Для того, чтобы определить точность этих часов, вызовите функцию **posix_trace_attr_getclockres**.

Элемент **posix_thread_id** - это идентификатор нити, генерирующей событие трассировки. Если элемент **posix_event_id** является одним из системных событий трассировки, и данное событие трассировки не связано ни с одной нитью, для элемента **posix_thread_id** устанавливается значение, равное нулю.

Атрибуты потока трассировки

Объект атрибутов потока трассировки **posix_trace_attr_t** состоит из перечисленных ниже атрибутов потока трассировки:

- Атрибут *genversion* идентифицирует источник и версию системы трассировки.
- Атрибут *tracename* представляет собой строку символов, определенную контроллером трассировки. Он служит для идентификации потока трассировки.
- Атрибут *creation-time* представляет время создания потока трассировки.
- Атрибут *clock-resolution* определяет точность часов, используемых для генерирования системного времени.
- Атрибут *stream-min-size* определяет минимальный размер потока трассировки, в байтах. Этот размер зарезервирован исключительно для событий трассировки.
- Атрибут *stream-full-policy* определяет стратегию, реализуемую при заполнении потока трассировки. Допустимые значения этого атрибута: POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL или POSIX_TRACE_FLUSH.
- Атрибут *max-data-size* определяет максимальный размер записи события трассировки, в байтах.
- Атрибут *inheritance* указывает на то, наследует ли вновь созданный поток трассировки трассировку родительского процесса. Допустимые значения этого атрибута: POSIX_TRACE_INHERITED или POSIX_TRACE_CLOSE_FOR_CHILD.
- Атрибут *log-max-size* определяет максимальный размер, в байтах, протокола трассировки, связанного с активным потоком трассировки. Остальные данные потока не включаются в этот размер.
- Атрибут *log-full-policy* определяет стратегию протокола трассировки, связанного с активным потоком трассировки. Допустимые значения: POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL или POSIX_TRACE_APPEND.

Определения типов событий трассировки

Типы системных и пользовательских событий трассировки определяются в файле заголовка <trace.h>. В этом разделе описаны типы событий трассировки с определениями.

Определения типов системных событий трассировки

Для интерпретации потока трассировки или протокола трассировки, процессу анализатора трассировки требуется информация о событиях трассировки и информация о системных событиях трассировки, предоставляющая сведения о вызове операций трассировки.

Вызов операций трассировки отслеживается следующими типами системных событий трассировки:

POSIX_TRACE_START
Отслеживание вызова операции начала трассировки.

POSIX_TRACE_STOP
Отслеживание вызова операции остановки трассировки.

POSIX_TRACE_FILTER
Отслеживание вызова операции изменения типа события трассировки.

Ниже перечислены типы системных событий трассировки, которые сообщают о событиях трассировки, записанных процедурами Posix Trace Library:

POSIX_TRACE_OVERFLOW
Начало переполнения трассировки.

POSIX_TRACE_RESUME
Окончание переполнения трассировки.

POSIX_TRACE_FLUSH_START
Начало операции очистки.

POSIX_TRACE_FLUSH_STOP
Окончание операции очистки.

POSIX_TRACE_ERROR
Ошибка трассировки.

События трассировки POSIX_TRACE_START и POSIX_TRACE_STOP указывают на время, истекшее с момента запуска потока трассировки.

Событие трассировки POSIX_TRACE_STOP со значением, равным нулю, указывает на вызов функции **posix_trace_stop**.

Событие трассировки POSIX_TRACE_STOP со значением, отличным от нуля, указывает на автоматическую остановку потока трассировки. Дополнительная информация приведена в описании функции **posix_trace_attr_getstreamfullpolicy**.

Событие трассировки POSIX_TRACE_FILTER указывает на то, что значение фильтра типов событий трассировки изменяется при запущенном потоке трассировки.

POSIX_TRACE_ERROR указывает на возникновение внутренней ошибки в системе трассировки.

Событие трассировки POSIX_TRACE_OVERFLOW выдается с системным временем, равным значению системного времени первого перезаписанного события трассировки. Это событие трассировки означает, что некоторые сгенерированные события трассировки утеряны.

Событие трассировки POSIX_TRACE_RESUME указывает на то, что система трассировки записывает события трассировки после переполнения.

Константа с именем события трассировки и константа **trace_event_id_t** определяют тип события трассировки. Данные события трассировки связаны с некоторыми из этих событий трассировки.

В таблице Системные события трассировки описаны стандартные системные события трассировки.

Таблица 83. Системные события трассировки

Имя события	Константа	Связанные данные	Тип данных
posix_trace_error	POSIX_TRACE_ERROR	Ошибка	**int**
posix_trace_start	POSIX_TRACE_START	event_filter	**trace_event_set_t**
posix_trace_stop	POSIX_TRACE_STOP	автоматически	**int**
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter, new_event_filter	**trace_event_set_t**
posix_trace_overflow	POSIX_TRACE_OVERFLOW	*Нет*	*Нет*
posix_trace_resume	POSIX_TRACE_RESUME	*Нет*	*Нет*
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	*Нет*	*Нет*
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	*Нет*	*Нет*

Определения типов пользовательских событий трассировки

Пользовательское событие трассировки POSIX_TRACE_UNNAMED_USEREVENT определяется в файле заголовка <trace.h>. При достижении ограничения TRACE_USER_EVENT_MAX, возвращается

пользовательское событие POSIX_TRACE_UNNAMED_USEREVENT, означающее, что приложение пытается зарегистрировать число событий, превышающее допустимое. Данных, связанных с этим пользовательским событием трассировки, нет.

Константа POSIX_TRACE_UNNAMED_USEREVENT является предопределенной для имени события трассировки **posix_trace_unnamed_userevent**.

Функции трассировки

В этом разделе описаны функции трассировки и различные роли трассировки, которые поддерживаются ими.

Структура интерфейса трассировки способствует оптимизации возможностей перемещения благодаря использованию данных трассировки непрозрачного типа.

Для того, чтобы настроить ресурсы для запуска потока трассировки процессом контроллера трассировки, используются функции, описанные в таблице ниже.

Функция	Назначение
posix_trace_attr_init	Инициализирует атрибуты
posix_trace_attr_destroy	Уничтожает атрибуты
posix_trace_attr_getgenversion	Возвращает версию потока трассировки
posix_trace_attr_getname	Возвращает имя трассировки
posix_trace_attr_setname	Задает имя трасировки
posix_trace_attr_getinherited	Возвращает стратегию наследования потока трассировки
posix_trace_attr_setinherited	Задает стратегию наследования потока трассировки
posix_trace_attr_setstreamfullpolicy	Задает полную стратегию потока
posix_trace_attr_getstreamfullpolicy	Возвращает полную стратегию потока
posix_trace_attr_setlogfullpolicy	Задает полную стратегию протокола потока трассировки
posix_trace_attr_getlogfullpolicy	Возвращает полную стратегию протокола потока трассировки
posix_trace_attr_setlogsize	Задает размер протокола потока трассировки
posix_trace_attr_setmaxdatasize	Задает максимальный размер данных пользовательского события трассировки
posix_trace_attr_setstreamsize	Задает размер потока трассировки
posix_trace_attr_getmaxusereventsize	Возвращает максимальный размер пользовательского события для заданной длины
posix_trace_create	Создает активный поток трассировки
posix_trace_create_withlog	Создает активный поток трассировки и связывает его с протоколом трассировки
posix_trace_flush	Копирует содержимок потока трассировки в связанный протокол трассировки потока трассировки
posix_trace_shutdown	Завершает работу потока трассировки
posix_trace_clear	Очищает поток трассировки и протокол трассировки
posix_trace_trid_eventid_open	Связывает идентификатор типа события трассировки с именем пользовательского события трассировки
posix_trace_eventid_equal	Сравнивает идентификаторы двух типов событий трассировки
posix_trace_eventid_get_name	Получает имя события трассировки из идентификатора типа события трассировки
posix_trace_eventtypelist_getnext_id, **posix_trace_eventtypelist_rewind**	Выполнение операции для всех элементов преобразования типов событий трассировки
posix_trace_eventset_add	Добавляет тип события трассировки в набор типов событий трассировки
posix_trace_eventset_empty	Очищает набор типов событий трассировки

Функция	Назначение
posix_trace_eventset_del	Удаляет тип события трассировки из набора типов событий трассировки
posix_trace_eventset_fill	Заполняет набор типов событий трассировки
posix_trace_eventset_ismember	Проверяет, включен ли данный тип события трассировки в набор типов событий трассировки
posix_trace_get_filter	Возвращает фильтр инициализированного потока трассировки
posix_trace_set_filter	Задает фильтр инициализированного потока трассировки
posix_trace_start	Запускает поток трассировки
posix_trace_stop	Останавливает поток трассировки
posix_trace_get_attr	Считывает информацию потока трассировки
posix_trace_get_status	Получает атрибуты трассировки или состояние трассировки

Для того, чтобы задать точку трассировки для трассируемого процесса, используются процедуры **posix_trace_event** и **posix_trace_eventid_open**. Эти процедуры предназначены для определения идентификаторов типов событий трассировки и вставки точек трассировки.

В приведенной ниже таблице показаны функции для получения информации из потока трассировки и протокола трассировки для процесса анализатора трассировки.

Функция	Назначение
posix_trace_attr_getname	Возвращает имя трассировки
posix_trace_attr_getgenversion	Считывает идентификационную информацию
posix_trace_attr_getcreatetime	Возвращает время создания потока трассировки
posix_trace_attr_getinherited	Возвращает стратегию наследования потока трассировки
posix_trace_attr_getstreamfullpolicy	Возвращает полную стратегию потока
posix_trace_attr_getlogfullpolicy	Возвращает полную стратегию протокола потока трассировки
posix_trace_attr_getmaxusereventsize	Возвращает максимальный размер пользовательского события для заданной длины
posix_trace_attr_getmaxsystemeventsize	Возвращает максимальный размер системного события трассировки
posix_trace_attr_getlogsize	Возвращает размер протокола потока трассировки
posix_trace_attr_getmaxdatasize	Возвращает максимальный размер данных пользовательского события трассировки
posix_trace_attr_getstreamsize	Возвращает размер потока трассировки
posix_trace_trid_eventid_open	Связывает идентификатор типа события трассировки с именем пользовательского события трассировки
posix_trace_eventid_equal	Сравнивает идентификаторы двух типов событий трассировки
posix_trace_eventid_get_name	Получает имя события трассировки из идентификатора типа события трассировки
posix_trace_eventtypelist_getnext_id, **posix_trace_eventtypelist_rewind**	Выполнение операции для всех элементов преобразования типов событий трассировки
posix_trace_open	Открывает протокол трассировки
posix_trace_close	Закрывает протокол трассировки
posix_trace_rewind	Повторно инициализирует протокол трассировки для считывания
posix_trace_get_attr	Считывает информацию потока трассировки
posix_trace_get_status	Считывает состояние потока трассировки
posix_trace_getnext_event	Считывает событие трассировки
posix_trace_timedgetnext_event, **posix_trace_trygetnext_event**	Возвращает событие трассировки

Подсистема tty

AIX - это многопользовательская операционная система, причем с ней могут работать пользователи как локальных, так и удаленных компьютеров. На уровне соединений эта возможность обеспечивается подсистемой tty.

Связью между терминалами и программами, работающими с ними, управляет интерфейс терминала. Примеры устройств tty приведены ниже:

- Модемы
- терминалы ASCII
- Системные консоли
- Последовательные принтеры
- Xterm или aixterm в X-Windows

В этом разделе рассматриваются следующие вопросы:

Задачи подсистемы TTY

Подсистема tty отвечает за:

- Управление передачей данных по асинхронной линии связи (скоростью передачи, размером символов и доступом к линии)
- Интерпретацию данных, включая распознавание специальных символов и определение языка
- Управление заданиями и доступом к терминалам с помощью модели управляющего терминала

Управляющий терминал обслуживает операции ввода-вывода группы процессов. Особый файл **tty** обеспечивает интерфейс для управляющего терминала. В реальной работе пользовательские программы редко явно открывают файлы терминалов, такие как **dev/tty5**. Эти файлы открываются командой **getty** или **rlogind**. После этого они применяются в качестве устройств стандартного ввода и вывода.

Подробная информация об управляющем терминале приведена в разделе **tty Special File** в книге *Справочник по файлам*.

Модули подсистемы tty

Для выполнения поставленных задач подсистема tty разбита на модули. Модуль отвечает за один из уровней передачи данных между компьютером и асинхронным устройством. Модули tty могут добавляться и удаляться динамически.

Подсистема tty поддерживает три основных типа модулей:

Драйверы tty

Драйверы tty, или аппаратные дисциплины, напрямую управляют аппаратным (для устройств tty) или псевдоаппаратным (для устройств pty) обеспечением. Драйверы занимаются обменом данными с адаптером, обеспечивая для вышестоящих модулей работу таких служб, как управление потоком и специальная семантика открытия порта.

Существуют следующие драйверы tty:

Драйвер	Описание
cxma	128-портовый асинхронный контроллер PCI.
cxpa	8-портовый асинхронный контроллер PCI.
lft	Низкоуровневый терминал. Имя этого терминала - **/dev/lft***Y*, где *Y* >= 0.
pty	драйвер псевдотерминала.
sa	2-портовый асинхронный адаптер PCI EIA-232.
sf	Универсальные асинхронные приемники/приемопередатчики (UART) на планаре системы.

Дополнительная информация приведена в разделе “Драйверы TTY” на стр. 808.

Дисциплины линии

Дисциплины линии предназначены для управления заданиями и интерпретации специальных символов. Они выполняют все необходимые преобразования входящих и исходящих потоков данных. Кроме того, дисциплины линии обрабатывают ошибки и отслеживают состояние драйвера tty.

Предусмотрены следующие дисциплины линии:

Функция	Описание
ldterm	Терминал
sptr	Принтер (команда **splp**)
slip	Линия связи SLIP (команда **slattach**)

Модули преобразования

Модули, или *дисциплины*, преобразования предназначены для обработки символов ввода и вывода.

Предусмотрены следующие модули преобразования:

Агент преобразования	Описание
nls	Поддержка национальных языков; этот модуль преобразует символы входящих и исходящих потоков в соответствии с таблицами, определенными для порта (см. описание команды **setmaps**)
lc_sjis и **uc_sjis**	Первичные и вторичные модули преобразования многобайтовых символов из SJIS (Shifted Japanese Industrial Standard) в AJEC (Advanced Japanese EUC Code) для дисциплины линии **ldterm**.

Дополнительная информация о модулях преобразования приведена в разделе “Модули преобразования” на стр. 807.

Структура подсистемы TTY

Подсистема tty основана на потоках. Это позволяет создать гибкую структуру, разбитую на модули, которая обладает следующими свойствами:

- Простота настройки: пользователи могут настраивать подсистему tty, добавляя и удаляя отдельные модули.
- Многократное использование модулей: например, один модуль дисциплины линии может применяться для различных устройств tty с разными конфигурациями.
- Простота добавления новых функций в подсистему терминалов.
- Возможность работы с различными устройствами через один интерфейс.

Поток tty состоит из следующих модулей:

- Начало потока, в котором обрабатываются пользовательские запросы. У всех потоков терминалов, независимо от дисциплины линии и драйвера, общее начало.

- Необязательный первичный модуль преобразования (например, **uc_sjis**), обрабатывающий входящие и исходящие данные перед дисциплиной линии.
- Дисциплина линии.
- Необязательный вторичный модуль преобразования (например, **lc_sjis**), обрабатывающий входящие и исходящие данные после дисциплины преобразования.
- Необязательный модуль преобразования символов (**nls**), обрабатывающий входящие и исходящие данные перед драйвером терминала.
- Конец потока: драйвер терминала.

Модули, связанные с поддержкой национальных языков, подключаются только при необходимости.

Для принтера модули поддержки национальных языков обычно отсутствуют, что существенно упрощает структуру потока.

Стандартные функции

Интерфейсы стандартных функций подсистемы tty описаны в файлах **/usr/include/sys/ioctl.h** и **/usr/include/termios.h**. Файл **ioctl.h**, применяемый всеми модулями, содержит структуру **winsize** и несколько команд ioctl. Файл **termios.h** содержит типы данных и функции, определенные в стандарте POSIX.

Эти функции объединены в группы по своему назначению и описаны ниже.

Функции управления аппаратным обеспечением

Для управления аппаратным обеспечением предназначены следующие функции:

Функция	Описание
cfgetispeed	Возвращает скорость получения данных в бодах
cfgetospeed	Возвращает скорость передачи данных в бодах
cfsetispeed	Задает скорость получения данных в бодах
cfsetospeed	Задает скорость передачи данных в бодах
tcsendbreak	Передает сигнал прерывания по асинхронной последовательной линии

Функции управления потоком

Для управления потоком предназначены следующие функции:

Функция	Описание
tcdrain	Ожидает завершения вывода
tcflow	Выполняет функции управления потоком
tcflush	Очищает указанную очередь

Функции для работы с терминалом

Для получения информации и управления терминалом предназначены следующие функции:

Функция	Описание
isatty	Определяет, является ли устройство терминалом
setcsmap	Считывает файл преобразования кодовых наборов и связывает его с устройством стандартного ввода
tcgetattr	Возвращает состояние терминала
tcsetattr	Задает состояние терминала
ttylock, **ttywait**, **ttyunlock** или **ttylocked**	Управляют функциями блокировки терминала
имя-терминала	Возвращает имя терминала

Функции изменения размера окна и терминала

Ядро сохраняет структуру **winsize** с информацией о размере текущего терминала или окна. Структура **winsize** содержит следующие поля:

Поле	Описание
`ws_row`	Число строк (символов) окна или терминала
`ws_col`	Число столбцов (символов) окна или терминала
`ws_xpixel`	Горизонтальный размер окна или терминала в пикселах
`ws_ypixel`	Вертикальный размер окна или терминала в пикселах

По принятому соглашению, значение 0 во всех полях структуры **winsize** означает, что она еще не была заполнена.

Функция	Описание
termdef	Запрашивает характеристики терминала.
TIOCGWINSZ	Возвращает размер окна. В качестве аргумента этой операции передается указатель на структуру **winsize**, в которую помещаются параметры текущего окна или терминала.
TIOCSWINSZ	Задает размер окна. В качестве аргумента этой операции передается указатель на структуру **winsize**, в которую помещаются новые параметры окна или терминала. Если переданная информация отличается от текущей, группе процессов терминала отправляется сигнал **SIGWINCH**.

Функции управления группами процессов

Для управления группой процессов предназначены следующие функции:

Функция	Описание
tcgetpgrp	Возвращает идентификатор интерактивной группы процессов
tcsetpgrp	Задает идентификатор интерактивной группы процессов

Операции изменения размера буфера

Следующие операции ioctl задают размер буферов ввода и вывода на терминал. В качестве аргумента этим операциям передается указатель на размер буфера.

Операции	Описание
TXSETIHOG	Задает максимальное число символов, которые могут быть получены и сохранены во внутренних буферах подсистемы tty до получения процессом. Значение по умолчанию равно 8192. При чтении символа с номером, на единицу превышающим заданное значение, в протокол заносится сообщение об ошибке, а буфер очищается. Размер буфера не должен быть слишком большим, поскольку буфер расположен в закрепленной памяти системы.
TXSETOHOG	Задает максимальное число символов, которые могут быть помещены в буфер для вывода. Значение по умолчанию равно 8192. После заполнения буфера символы перестают выводиться. Размер буфера не должен быть слишком большим, поскольку буфер расположен в закрепленной памяти системы.

Синхронизация

Синхронизация подсистемы tty обеспечивается модулем STREAMS. Модули потоков tty поддерживают синхронизацию на уровне пар очередей. Такая синхронизация позволяет разделить обработку на два параллельных потока.

Информация, связанная с данной:

rlogind

setmaps

stty

xdm

eucioctl.h

lft

pty

Файл setmaps

termios.h

файл tty

Модуль дисциплины линии (ldterm)

Ldterm - это стандартная дисциплина линии для терминалов.

Она соответствует стандарту POSIX и совместима с интерфейсом BSD. Последний поддерживается только для обеспечения совместимости с устаревшими приложениями. Для обеспечения переносимости приложений настоятельно рекомендуется разрабатывать новые приложения на основе стандарта POSIX.

В этом разделе описаны возможности протокола **ldterm**. Дополнительная информация о работе с **ldterm** приведена в разделе "termios.h File" руководства *Справочник по файлам* .

Параметры терминала

Параметры ввода-вывода различных терминалов задаются в структуре **termios**, описанной в файле **termios.h**. Ниже перечислены некоторые (но не все) элементы структуры **termios**:

`tcflag_t c_iflag`
: Режимы ввода

`tcflag_t c_oflag`
: Режимы вывода

`tcflag_t c_cflag`
: Режимы управления

`tcflag_t c_lflag`
: Режимы локали

`cc_t c_cc[NCCS]`
: Управляющие символы.

В файле **termios.h** определены целочисленные типы без знака `tcflag_t` и `cc_t`. Также в файле **termios.h** определен символ **NCCS**.

Управление группами процессов в рамках сеанса (управление заданиями)

Управляющий терминал выделяет одну группу процессов в рамках сеанса, с которым он связан, и объявляет ее основной (интерактивной) группой процессов. Все остальные группы процессов данного сеанса объявляются фоновыми. Основная группа процессов играет особую роль при обработке сигналов.

Процессы интерпретации команд с поддержкой управления заданиями, например, оболочка Korn (команда **ksh**) и оболочка C (команда **csh**), могут выделить терминал нескольким *заданиям*, или группам процессов, путем объединения этих процессов в одну группу и назначения терминала этой группе. Процесс может установить или опросить основную группу процессов терминала, если у этого процесса есть соответствующие права доступа. Драйвер терминала содействует разбиению заданий на группы, запрещая доступ к терминалу процессам, не входящим в основную группу.

Управление доступом к терминалу

Если процесс, не входящий в основную группу управляющего терминала, пытается считать данные с терминала, то в группу этого процесса передается сигнал **SIGTTIN**. Если же данный процесс игнорирует или блокирует сигнал **SIGTTIN** или если группа этого процесса недоступна, то запрос на чтение возвращает код -1, присваивает глобальной переменной **errno** значение **EIO** и не передает сигнал.

Если процесс, не входящий в основную группу управляющего терминала, пытается записать данные на терминал, то в группу этого процесса передается сигнал **SIGTTOU**. Однако управление сигналом **SIGTTOU** зависит от флага **TOSTOP**, определенного в поле `c_lflag` структуры **termios**. Если флаг **TOSTOP** сброшен, а также если флаг **TOSTOP** установлен, но процесс игнорирует или блокирует сигнал **SIGTTOU**, этому процессу разрешается запись на терминал, а сигнал **SIGTTOU** не передается. Если флаг **TOSTOP** установлен, группа процесса, запросившего запись, недоступна, а процесс не игнорирует и не блокирует сигнал **SIGTTOU**, то запрос на запись возвращает код -1, присваивает глобальной переменной **errno** значение **EIO** и не передает сигнал.

Некоторые функции, устанавливающие параметры терминала (**tcsetattr**, **tcsendbreak**, **tcflow** и **tcflush**), обрабатываются так же, как и запросы на запись, за исключением того, что флаг **TOSTOP** игнорируется. Иными словами, они обрабатываются так же, как и запросы на запись при установленном флаге **TOSTOP**.

Чтение данных и обработка ввода

Существует два основных типа обработки ввода. Выбор конкретного типа определяется тем, находится ли файл устройства для терминала в стандартном или же в нестандартном режиме. Кроме того, при обработке вводимых символов учитываются значения полей `c_iflag` и `c_lflag`. В ходе обработки возможен *эхоповтор*, т.е. введенные символы могут немедленно передаваться обратно на терминал, с которого они поступили. Эхоповтор полезен при работе в дуплексном режиме.

Запрос на чтение может обрабатываться одним из двух способов в зависимости от того, установила ли функция **open** или **fcntl** флаг **O_NONBLOCK**. Если флаг **O_NONBLOCK** сброшен, то запрос на чтение блокируется до тех пор, пока данные не станут доступными или не будет получен сигнал. Если флаг **O_NONBLOCK** установлен, то запрос на чтение будет выполнен одним из трех способов:

- Если данных достаточно для полного выполнения запроса, то обработка запроса на чтение завершается успешно с возвратом нужного числа считанных байт.
- Если данных недостаточно для полного выполнения запроса, то обработка запроса на чтение завершается успешно с возвратом всех байт, которые удалось считать.
- Если данные недоступны, то запрос на чтение возвращает код -1 и присваивает глобальной переменной **errno** значение **EAGAIN**.

Доступность данных зависит от режима обработки ввода (стандартный или нестандартный). Переключение между стандартным и нестандартным режимами осуществляется с помощью команды **stty**.

Обработка ввода в стандартном режиме

В стандартном режиме обработки ввода (в поле `c_lflag` структуры **termios** установлен флаг **ICANON**) ввод с терминала обрабатывается построчно. Ограничителем строки считается любой из следующих символов: символ начала строки (ASCII LF), символ конца файла (EOF) или символ конца строки (EOL). Это означает, что программа, отправившая запрос на чтение, блокируется до тех пор, пока не будет напечатана целая

строка или пока не поступит сигнал. Кроме того, запрос на чтение возвращает не более одной строки, независимо от того, сколько символов было указано в запросе. Однако необязательно считывать только целые строки. В запросе на чтение может быть указано любое число символов, и потери информации не произойдет. Обработка символов удаления выполняется в ходе ввода.

Символ СТИРАНИЯ
Удаляет последний введенный символ (по умолчанию - клавиша Backspace).

Символ удаления слова (WERASE)
Удаляет последнее слово, введенное в текущей строке, но оставляет предшествующие этому слову пробелы и символы табуляции (по умолчанию - комбинация клавиш Ctrl-W).

(*Словом* считается последовательность символов, не содержащая пробелов и символов табуляции.) Ни ERASE, ни WERASE не удаляют символы из предыдущих строк.

Символ ЗАВЕРШЕНИЯ ПРОЦЕССА
Удаляет всю введенную строку и (необязательно) печатает символ начала строки (по умолчанию - комбинация клавиш Ctrl-U).

Все эти символы обрабатываются по нажатии клавиш, причем независимо от нажатий клавиш Backspace и Tab.

Символ REPRINT
Печатает символ начала строки, а затем те символы из предыдущей строки, которые не были считаны (по умолчанию - комбинация клавиш Ctrl-R).

Повторная печать выполняется также автоматически, если символы, которые в обычных условиях были бы удалены с экрана, были выведены на него программно. Повторная печать символов выполняется так же, как эхоповтор. Следовательно, если флаг **ECHO** в поле `c_lflag` структуры **termios** сброшен, повторная печать не выполняется. Для ввода символов ERASE и KILL следует поставить перед ними символ \ (обратная косая черта): в этом случае сам символ \ не будет обрабатываться. Символы ERASE, WERASE и KILL можно переопределять.

Обработка ввода в нестандартном режиме

В нестандартном режиме обработки ввода (в поле `c_lflag` структуры **termios** установлен флаг **-ICANON**), вводимые байты не разбиваются на строки, а символы удаления слов и строк игнорируются.

MIN Минимальное число байт, которые необходимо принять для того, чтобы запрос на чтение был обработан успешно

TIME Таймер с шагом 0,1 с, применяемый для отсчета тайм-аута и при передаче данных короткими порциями

Значения элементов MIN и TIME массива `c_cc` определяют способ обработки принятых данных. Значения MIN и TIME можно установить с помощью команды **stty**. Допустимы значения MIN и MAX от 0 до 265. Четыре возможных комбинации параметров MIN и TIME и их сочетания описаны в следующих разделах.

Case A: MIN0, TIME0

В этом случае TIME задает интервал приема следующего байта: таймер активизируется после приема первого байта и сбрасывается при приеме каждого следующего байта. Если до срабатывания таймера считано MIN байт, то обработка запроса на чтение завершается успешно. Если до срабатывания таймера принято меньше MIN байт, пользователю передаются все символы, принятые до этого момента. Если операция чтения была прервана по таймеру, то будет возвращен хотя бы один байт. (Если ни одного байта не принято, то таймер не активизируется.) Операция чтения блокируется до активизации механизмов MIN и TIME после приема первого байта или же до приема сигнала.

Case B: MIN0, TIME = 0

В этом случае существенно только значение MIN, таймер не включается (значение TIME равно 0). Ожидающий запрос на чтение не завершается (блокируется) до получения MIN байт или до приема сигнала. Программа, работающая по такой схеме с терминалом, обрабатывающим записи, может ожидать выполнения запроса на чтение бесконечно долго.

Case C: MIN = 0, TIME0

В этом случае, так как значение MIN равно 0, TIME не может задавать интервал приема следующего байта. В этом случае он служит в качестве таймера чтения, который активируется во время обработки запроса на чтение. В данном случае это значение задает интервал приема всех данных. Таймер включается в момент начала обработки запроса. Обработка запроса на чтение завершается успешно после приема одного байта или после срабатывания таймера. Обратите внимание, что если таймер сработал, не будет возвращено ни одного байта. Если таймер не сработал, обработка запроса может завершиться успешно только в случае, если байт будет принят. Таким образом, применение этой схемы позволяет избежать "зависания" программы на неограниченное время в ожидании одного байта. Если после начала обработки запроса на чтение в течение TIME * 0,1 секунды не было принято ни одного байта, запрос на чтение возвращает значение 0 - данные не считаны.

Вариант D: MIN = 0, TIME = 0

В этом случае в ответ на запрос возвращается либо запрошенное число байт, либо все доступные байты, в зависимости от того, какое из этих чисел меньше, без ожидания ввода дополнительных данных. Если доступных символов нет, возвращается значение 0 - данные не считаны.

Случаи A и B подходят для обработки запросов на серийное чтение данных, например, от программ передачи файлов, если для работы программа должна за один запрос получить некоторое минимальное число символов. Это число указывается в переменной MIN. В случае A для обеспечения надежности включается таймер. В случае B таймер выключен.

Случаи C и D подходят для обработки ограниченной посимвольной передачи данных. Они применяются при работе с приложениями вывода данных на экран, которым нужно знать, содержался ли в очереди ввода хотя бы один символ, перед обновлением экрана. В случае C устанавливается таймер. В случае D таймер выключен. Случай D отличается более низкой производительностью, но его следует предпочесть запросам на чтение с установкой флага **O_NONBLOCK**.

Запись данных и обработка вывода

При записи одного или нескольких символов они передаются на терминал сразу после вывода на экран ранее записанных символов. (Эффект эхоповтора достигается путем помещения введенных символов в очередь вывода сразу же при вводе.) Если данные поступают из процесса быстрее, чем они могут выводиться на экран, обработка процесса приостанавливается до тех пор, пока длина очереди вывода не станет меньше заранее заданного значения. После этого выполнение программы возобновляется.

Управление модемом

Если в поле `c_cflag` структуры **termios** установлен флаг **CLOCAL**, то соединение не зависит от содержимого строк состояния модема. Если же флаг **CLOCAL** сброшен, то выполняется анализ строк состояния модема. В обычной ситуации функция **open** ожидает установки соединения с модемом. Если же установлен флаг **O_NONBLOCK** или **CLOCAL**, то возврат из функции open выполняется немедленно, без ожидания установки соединения.

Если флаг **CLOCAL** не установлен в поле `c_cflag` структуры **termios**, и интерфейс терминала обнаружил разрыв соединения модема с управляющим терминалом, то управляющему процессу, связанному с терминалом, отправляется сигнал **SIGHUP**. Если реакция на этот сигнал не была изменена, он вызывает завершение процесса. Если сигнал **SIGHUP** игнорируется или перехватывается, то в ответ на все

последующие запросы на чтение вплоть до закрытия терминала передается индикатор конца файла. Все эти запросы на чтение возвращают код -1 и присваивают глобальной переменной **errno** значение **EIO** вплоть до закрытия устройства.

Закрытие файла устройства для терминала

Последний процесс, закрывающий файл устройства для терминала, отправляет весь вывод в устройство и аннулирует весь ввод. Если установлен флаг **HUPCL** в поле `c_cflag` структуры **termios** и порт связи поддерживает функцию разъединения, терминал выполняет разъединение.

Модули преобразования

Процедуры преобразования представляют собой дополнительные компоненты; при необходимости они могут применяться в потоке терминала.

Как правило, эти процедуры служат для обеспечения работы системы с разными языками и выполняют преобразования символов.

Предусмотрены следующие модули преобразования:

- Модуль **nls**
- Модули **uc_sjis** и **lc_sjis**.

Модуль NLS

Модуль **nls** - это модуль преобразования нижнего уровня, который можно встраивать в поток данных терминала ниже уровня дисциплины линии. Модуль **nls** обеспечивает преобразование символов при вводе и выводе на нестандартных терминалах (не поддерживающих основной кодовый набор системы - ISO 8859).

Правила преобразования указаны в двух файлах преобразования в каталоге **/usr/lib/nls/termmap**. Файлы с расширением **.in** содержат правила преобразования символов, вводимых с клавиатуры. Файлы с расширением **.out** содержат правила преобразования символов, выводимых на дисплей. Формат этих файлов описан в разделе "**setmaps** file format" книги *Справочник по файлам* .

Модули SJIS

Модули преобразования **uc_sjis** и **lc_sjis** также можно записывать в поток `tty`. Они обеспечивают обработку определенных кодовых наборов: например, выполняют преобразование многобайтовых символов между японским стандартом SJIS и расширенным кодовым набором EUC для японского языка (AJEC) (это преобразование поддерживается протоколом линии передачи данных). Эти модули нужны, когда пользовательский процесс и терминал работают с кодовым набором IBM-943.

KAJEC - это японская реализация метода кодирования EUC (Расширенный код UNIX), который позволяет работать одновременно с символами ASCII, фонетическими символами каны и идеографическими символами кандзи. AJEC представляет собой надмножество UJIS (Японского промышленного стандарта для UNIX), являющегося наиболее распространенной реализации EUC для японского языка.

Данные на японском языке могут включать в себя символы из четырех кодовых наборов:

Кодовый набор	Символы
ASCII	Латинские буквы, цифры, знаки пунктуации и управляющие символы
JIS X0201	Фонетические символы Кана
JIS X0208	Идеографические символы кандзи
JIS X0212	Дополнительные символы кандзи.

В AJEC применяются все четыре кодовых набора. В SJIS применяются лишь наборы ASCII, JIS X0201 и JIS X0208. В связи с этим модули **uc_sjis** и **lc_sjis** выполняют следующие преобразования:

- Все символы SJIS - в символы AJEC
- Символы AJEC кодировок ASCII, JIS X0201 и JIS X0208 в символы SJIS
- Символы AJEC кодировки JIS X0212 в неопределенные символы SJIS.

Модули **uc_sjis** и **lc_sjis** всегда применяются совместно. Модуль преобразования верхнего уровня **uc_sjis** записывается в поток терминала над уровнем линии передачи данных; а модуль **lc_sjis** - под этим уровнем. Модули **uc_sjis** и **lc_sjis** можно записывать в поток терминала автоматически с помощью команды **setmaps** или функции **setcsmap**. Кроме того, ими можно управлять с помощью операций ioctl EUC, информация о которых приведена в описании файла **eucioctl.h** в книге *Справочник по файлам* .

Драйверы TTY

Драйвер tty - это драйвер потока, управляющий физическим соединением с терминалом.

В зависимости от соединения, есть три типа драйверов tty: драйверы асинхронных линий, драйвер pty и драйвер LFT.

Драйверы асинхронных линий

Драйверы асинхронных линий предназначены для обслуживания устройств (обычно текстовых терминалов), напрямую подключенных к системе с помощью асинхронных линий связи, в том числе через модем.

Драйверы асинхронных линий предоставляют интерфейс для управления физической линией:

- Драйвер **cxma** поддерживает карту 128-портового адаптера PCI.
- Драйвер **cxpa** поддерживает карту 8-портового адаптера PCI.
- Драйвер **sf** поддерживает встроенные порты на планаре системы.
- Драйвер **sa** поддерживает карту 2-портового адаптера PCI.

Драйверы асинхронных линий отвечают за настройку параметров, таких как скорость передачи, размер символов и проверка четности. Пользователь может изменять эти параметры с помощью поля `c_cflag` структуры **termios**.

Кроме того, драйверы асинхронных линий выполняют следующие функции:

- Функция аппаратного и программного управления потоком, или дисциплина управления передачей, предотвращает переполнение буфера. Пользователь может управлять этой функцией с помощью поля `c_iflag` структуры **termios** (программное управление передачей) и поля `x_hflag` структуры **termiox** (аппаратное управление передачей).
- Дисциплина открытия определяет процедуру установления соединения. Эта функция настраивается с помощью поля `x_sflag` структуры **termiox**.

Драйвер псевдотерминала

Драйвер псевдотерминала (pty) предназначен для обслуживания специальных терминалов, таких как X-терминалы и удаленные системы, подключенные по сети.

Драйвер pty передает данные приложения процессу сервера, организуя второй поток. Процесс сервера, выполняющийся в пользовательском пространстве, обычно является демоном, таким как **rlogind** или **xdm**. Он управляет соединением с терминалом.

В потоке, со стороны пользователя или сервера, могут присутствовать дополнительные модули.

Группа библиотек

В некоторых средах программирования полезно предоставить различным процессам доступ к общей библиотеке по одному и тому же виртуальному адресу.

Поскольку системный загрузчик AIX управляет общими библиотеками динамически, обеспечить совпадение адресов в общем случае нельзя. Группы библиотек позволяют загружать общие библиотеки по одному и тому же виртуальному адресу для нескольких процессов.

Системный загрузчик загружает общие библиотеки в несколько глобальных областей памяти для общих библиотек. Одна из этих областей называется областью текста общих библиотек, в ней хранятся исполняемые инструкции загруженных общих библиотек. Область текста общих библиотек отображается в каждом процессе по одному и тому же виртуальному адресу. Другая область - область данных общих библиотек. Эта область содержит данные общих библиотек. Поскольку данные общих библиотек доступны для чтения и записи, каждому процессу выделяется отдельная закрытая область, в которую помещается копия глобальной области данных общих библиотек. Эта закрытая область отображается в каждом процессе по одному и тому же виртуальному адресу.

Так как глобальные области памяти общих библиотек отображаются в каждом процессе по одному и тому же виртуальному адресу, общие библиотеки в большинстве случаев загружаются в область памяти, расположенную по одному и тому же виртуальному адресу. Исключение составляет случай, когда в системе загружено несколько версий одной и той же общей библиотеки. Это происходит, если используемая общая библиотека или одна из общих библиотек, от которых она зависит, была изменена. В этом случае загрузчик должен создать новую версию измененной общей библиотеки и всех общих библиотек, зависящих от нее. Заметим, что все общие библиотеки зависят от *пространства имен ядра*. *Пространство имен ядра* содержит все системные вызовы, определенные в ядре, и его можно изменить во время любой динамической загрузки или выгрузки расширения ядра. Если системный загрузчик создал новую версию общей библиотеки, то она должна размещаться отдельно от глобальных сегментов общей библиотеки. Поэтому процессы, работающие с новой версией, и процессы, работающие со старой, должны обращаться к различным областям памяти по различным виртуальным адресам.

Группа библиотек - это несколько общих библиотек, загруженных в системе. Полный набор общих библиотек, загруженных в системе, называется *глобальной группой библиотек*. Глобальная группа может подразделяться на несколько пользовательских групп библиотек. В пользовательской группе библиотек содержится по одной версии каждой общей библиотеки. В процессах можно указывать группу библиотек. Если в процессе указана группа библиотек, то этот процесс будет работать с общими библиотеками, входящими в эту группу. Если несколько процессов связаны с одной и той же группой библиотек, все они работают с одним и тем же набором общих библиотек. Так как в группе библиотек не может быть двух версий одной библиотеки, все процессы, связанные с данной группой библиотек, работают с одной и той же версией каждой библиотеки, а кроме того, общие библиотеки этих процессов загружаются под одному и тому же виртуальному адресу.

Работа с группами библиотек

Если процесс должен обращаться к группе библиотек, то это необходимо указать во время выполнения. После этого указанная группа библиотек подключается к процессу, и процесс может с ней работать до своего завершения. Если процесс, при запуске которого была указана группа библиотек, выполняет системный вызов **exec**, системный загрузчик выполняет следующие действия:

Выполняет поиск группы библиотек, а если она не найдена, создает ее
Загрузчик проверяет права доступа для данной группы библиотек с целью выяснить, может ли этот процесс работать с ней. Если у процесса нет нужных прав доступа к группе библиотек (на чтение или запись), то процесс не будет связан ни с какой группой библиотек. Если же у процесса есть требуемые права доступа, система выполняет поиск указанной группы в списке существующих групп библиотек. Если нужная группа процессов не найдена, а у процесса есть требуемые права доступа, группа создается. Если же у процесса недостаточно прав доступа для создания группы процессов, системный вызов **exec** завершается неудачно и возвращается код ошибки.

Ограничивает область поиска данной группой библиотек
Если процесс обращается к любой общей библиотеке, входящей в группу, то используется версия библиотеки из данной группы. Процесс всегда работает с версиями общих библиотек, входящими в группу, независимо от того, если ли в глобальной группе библиотек другие версии.

Добавляет общие библиотеки в группу
Если процессу требуется библиотека, не входящая в данную группу, загрузчик загружает самую свежую версию этой библиотеки в образ процесса. Если у процесса есть соответствующие права доступа, эта версия библиотеки также добавляется в данную группу библиотек. Если у процесса нет прав на добавление записей, то системный вызов **exec** завершается неудачно и возвращается код ошибки.

Общие библиотеки можно загружать и явно, с помощью системного вызова **load( )**. При явной загрузке общей библиотеки данные для ее модулей, как правило, помещаются по адресу текущего прерывания процесса, если процесс 32-разрядный. Для 64-разрядных процессов данные модулей помещаются в закрытую область памяти процесса. Если процесс связан с группой библиотек, системный загрузчик помещает эти данные в область данных общей библиотеки. Виртуальный адрес явно загруженного данным процессом модуля тот же, что и для всех процессов, загрузивших его. Если у процесса есть соответствующие права доступа, библиотека также добавляется в данную группу библиотек. Если у процесса нет прав на добавление записей, то системный вызов **load** завершается неудачно и возвращается код ошибки.

Группу библиотек можно связать с любым обычным файлом. Важно отметить, что группа данных связывается с самим файлом, а НЕ с его (полным) именем. Режим доступа (права доступа) к файлу определяет допустимые действия над группой библиотек. Ниже перечислены права доступа к файлу, связанному с группой библиотек, и допустимые операции над группой библиотек:

- Если процесс может считывать данные из файла, то процесс может при запуске указать соответствующую группу библиотек, чтобы ограничить диапазон поиска.
- Если процесс может записывать данные в файл, то процесс может добавлять общие библиотеки в группу библиотек и создавать группу библиотек, связанную с файлом.

Если процесс пытается создать группу библиотек или добавить в нее библиотеку, не обладая соответствующими правами доступа, то текущая операция (**exec** или **load**) завершается неудачно, после чего возвращается код ошибки.

Группы библиотек перечислены в поле **LIBPATH**. Поле **LIBPATH** - это список разделенных двоеточиями (:) имен каталогов, в которых могут находиться общие библиотеки. Данные **LIBPATH** поступают либо из переменной среды **LIBPATH**, либо из строки **LIBPATH**, содержащейся в загрузочном разделе исполняемого файла. Если первое полное имя в списке **LIBPATH** - это имя обычного файла, то выбирается группа библиотек, связанная с этим файлом. Пример:

- Если `/etc/loader_domain/OOdomain_1` - это обычный файл, а переменной среды **LIBPATH** присвоено значение

  ```
  /etc/loader_domain/OOdomain_1:/lib:/usr/lib
  ```

  то процессы будут создавать и применять группу библиотек, связанную с файлом `/etc/loader_domain/OOdomain_1`.
- Если `/etc/loader_domain/OOdomain_1` - обычный файл, то следующая команда создаст программу `ldom`:

  ```
  cc -o ldom ldom.c -L/etc/loader_domain/OOdomain_1
  ```

Имя /etc/loader_domain/OOdomain_1 станет первой записью в строке **LIBPATH** в загрузочном разделе файла ldom. При выполнении программы ldom она создаст и будет использовать группу библиотек, связанную с файлом /etc/loader_domain/OOdomain_1.

Создание и удаление групп библиотек

Группа библиотек создается при первом обращении к ней процесса, обладающего соответствующими правами доступа. Организация доступа к группе библиотек зависит от организации доступа к связанному с ней обычному файлу. Ответственность за управление обычными файлами, связанными с нужными группами библиотек, ложится на разработчиков приложений. Напоминаем, что группы библиотек связываются с файлами, а НЕ с их именами. Рассмотрим следующие примеры:

- В списке **LIBPATH** приложения apl задается группа библиотек domain01. После этого приложение apl выполняется. Текущий рабочий каталог - /home/user1, в нем содержится обычный файл domain1, в который приложение apl может записывать данные. Создается новая группа библиотек, связанная с файлом /home/user1/domain01. После этого apl выполняется еще раз. Теперь рабочий каталог - /home/user2, и в этом каталоге тоже есть обычный файл domain01, в который приложение apl может записывать данные. Создается новая группа библиотек, связанная с файлом /home/user1/domain02.
- В списке **LIBPATH** приложения apl указывается группа библиотек /etc/1_domain/domain01. После этого приложение apl выполняется. /etc/1_domain/domain01 - это обычный файл, в который apl может записывать данные. Создается новая группа библиотек, связанная с файлом /etc/1_domain/domain01.

  /home/user1/my_domain - это символьная связь с файлом /etc/1_domain/domain01.

  В списке **LIBPATH** приложения ap2 указывается группа библиотек /home/user1/my_domain. После этого приложение ap2 выполняется. Системный загрузчик обнаруживает, что /home/user1/my_domain указывает на тот же файл, что и /etc/1_domain/domain01. С файлом /etc/1_domain/domain01 уже связана группа библиотек; следовательно, приложение ap2 будет работать с этой группой.
- В списке **LIBPATH** приложения apl указывается группа библиотек /etc/1_domain/domain01. После этого приложение apl выполняется. /etc/1_domain/domain01 - это обычный файл, в который apl может записывать данные. Создается новая группа библиотек, связанная с файлом /etc/1_domain/domain01.

  Файл /etc/1_domain/domain01 удаляется и вновь создается как обычный файл.

  Приложение apl выполняется еще раз. Теперь уже нельзя обратиться к обычному файлу, связанному с исходной группой библиотек /etc/1_domain/domain01. Поэтому создается новая группа библиотек, связанная с файлом /etc/1_domain/domain01.

Группы библиотек являются динамическими структурами. В уже существующую группу библиотек можно добавлять общие библиотеки, можно также удалять их. Общая библиотека добавляется в группу, когда процесс, связанный с этой группой, обращается к общей библиотеке, еще не входящей в эту группу. Разумеется, библиотека добавляется в группу только при условии, что у процесса есть соответствующие права доступа.

Для каждой общей библиотеки, входящей в группу, существует отдельный счетчик процессов, работающих с библиотекой. В нем хранится информация о том, сколько процессов, связанных с группами библиотек, работают с данной общей библиотекой. Когда этот счетчик обнуляется, общая библиотека удаляется из группы.

API управления данными

Интерфейс прикладных программ управления данными (DMAPI) реализован в AIX в соответствии со стандартом X/Open "System Management: Data Storage Management (XDSM) API", опубликованном Open Group.

С помощью DMAPI разработчики программного обеспечения могут создавать прикладные программы управления данными, используя набор функций и семантику, отсутствующие в POSIX-совместимых системах. Эти функции недоступны непосредственно пользователям. Полная документация по DMAPI приведена в разделе публикаций на Web-сайте Open Group.

В AIX реализована общая функциональность DMAPI. Уровень поддержки необязательных функций зависит от используемой файловой системы и документирован в разделах, посвященных конкретным файловым системам.

Основное назначение DMAPI - обеспечить поддержку отдельного продукта в конкретной файловой системе. Использование DMAPI для нескольких продуктов разных производителей в одной и тоже файловой системе не запрещено, однако не рекомендуется. В отношении диспетчеризации сообщений DMAPI полностью поддерживает различные продукты в разных файловых системах, но при этом действуют следующие ограничения:

- Невозможно управлять одним и тем же событием для одного и того же объекта из нескольких сеансов.
- Сообщения о событиях и их очереди привязаны к сеансам; событие не может быть явным образом предназначено конкретному процессу.
- Если ни один сеанс не зарегистрировал событие, отличное от события монтирования, то DMAPI не будет генерировать событие, и работа процесса будет продолжена, как если бы событие вообще не произошло. Если ни один сеанс не зарегистрировал событие монтирования, которое всегда включено, то DMAPI сбросит событие и запретит монтирование файловой системы.

В AIX DMAPI реализован на абстрактном уровне, что позволяет определять индивидуальные уровни его поддержки и степень полноты реализации в различных файловых системах. В файловой системе JFS поддержка DMAPI отсутствует. Детали его реализации в файловой системе JFS2, ограничения и другие опции в соответствии со стандартом X/Open описаны в разделе Особенности реализации DMAPI в JFS2.

Функция **dm_init_service** возвращает 0, если DMAPI AIX инициализируется правильно, или -1, если инициализация не выполняется. Вызов любых функций DMAPI после ошибки инициализации также приведет к ошибке.

Следующие необязательные функции DMAPI не поддерживаются в AIX:

- **dm_downgrade_right**
- **dm_upgrade_right**
- **dm_obj_ref_*** family
- **dm_pending**

Другие необязательные функции могут не поддерживаться в конкретных файловых системах. Дополнительная информация приведена в документации по DMAPI для конкретных файловых систем.

Когда прикладная программа управления данными запрашивает блокировку для ожидания права доступа к какому-либо объекту, эту блокировку прервать невозможно.

В AIX разрешены множественные неперекрывающиеся постоянные управляемые области. Управляемые области могут принадлежать только обычным файлам. Возможность переупорядочивания или сращивания управляемых областей зависит от файловой системы.

Если ни один сеанс не зарегистрировался для получения события, для которого задан объект и произошло действие, требуемое для срабатывания события, то AIX не будет генерировать событие, и работа процесса будет продолжена, как если бы событие вообще не произошло.

Запуск функции **dm_set_eventlist** приводит к созданию постоянного списка событий, хранящегося вместе с объектом. Если список событий был ранее задан для всей файловой системы, и в этом списке присутствует событие, добавляемое в список событий объекта, то при генерации событий будет использоваться список файловой системы. В таком случае, если это событие будет удалено из списка файловой системы, вступит в силу список событий объекта.

Когда процесс, генерирующий событие, блокируется в ожидании ответа от приложения управления данными, эту блокировку прервать невозможно.

В AIX реализован механизм доставки асинхронных сообщений с ограниченной надежностью. Число ожидающих доставки асинхронных сообщений ограничено объемом физической или виртуальной памяти системы. Если число сообщений превысит возможности памяти, то недоставленные асинхронные сообщения будут утеряны. Способ доставки асинхронных сообщений о событиях пространства имен зависит от файловой системы.

В AIX DM_SESSION_INFO_LEN равен 256, а DM_ATTR_NAME_SIZE равен 8.

Если функция интерфейса DMAPI, возвращающая данные в буфер пользователя и записывающая в переменную размер этих данных, заканчивает работу с ошибкой, отличной от E2BIG, то как данные самого буфера, так и значение переменной размера будут не определены. При возникновении такой ошибки данные буфера использовать нельзя. Если функция заканчивает работу с ошибкой E2BIG, то значение переменной размера будет содержать требуемый размер буфера, и в этом случае функцию можно вызвать повторно с увеличенным размером буфера.

Особенности реализации DMAPI в JFS2

Примечание: Внутренние моментальные снимки не могут быть использованы с файловой системой, управляемой DMAPI.

В следующей таблице перечислены особенности реализации и функционирования DMAPI AIX в JFS2.

В JFS2 функция **dm_get_config** возвращает следующие значения:

DM_CONFIG_BULKALL
Поддерживается

DM_CONFIG_LEGACY
Поддерживается

DM_CONFIG_PERS_ATTRIBUTES
Поддерживается

DM_CONFIG_PERS_EVENTS
Поддерживается

DM_CONFIG_PERS_INHERIT_ATTRIBS
Поддерживается

DM_CONFIG_PERS_MANAGED_REGIONS
Поддерживается

DM_CONFIG_PUNCH_HOLE
Поддерживается

DM_CONFIG_WILL_RETRY
Поддерживается

DM_CONFIG_CREATE_BY_HANDLE
Не поддерживается

DM_CONFIG_LOCK_UPGRADE
Не поддерживается

DM_CONFIG_OBJ_REF
Не поддерживается

DM_CONFIG_PENDING
Не поддерживается

DM_CONFIG_DTIME_OVERLOAD
TRUE

DM_CONFIG_MAX_ATTR_ON_DESTROY
128

DM_CONFIG_MAX_ATTRIBUTE_SIZE
4072

DM_CONFIG_MAX_HANDLE_SIZE
32

DM_CONFIG_MAX_MANAGED_REGIONS
167

DM_CONFIG_MAX_MESSAGE_DATA
65536

DM_CONFIG_TOTAL_ATTRIBUTE_SPACE
4072

В JFS2 все атрибуты управления данными размещены в одной и той же области памяти. Поэтому размер любого атрибута не может превышать DM_CONFIG_MAX_ATTRIBUTE_SIZE. Более того, сумма размеров всех атрибутов управления данными, связанных с объектом, также не может превышать DM_CONFIG_MAX_ATTRIBUTE_SIZE.

Помимо необязательных функций интерфейса, не поддерживаемых в AIX, в JFS2 не поддерживаются необязательные события DMAPI cancel и debut, а также функции **dm_getall_dmattr**, **dm_create_by_handle** и **dm_symlink_by_handle**.

Вследствие особенностей реализации поддержки расширенных атрибутов в JFS2 вызов функции **dm_set_region** приводит к изменению ctime файла. В JFS2 не выполняется переупорядочение и слияние управляемых областей.

Асинхронные события пространства имен генерируются в JFS2 для всех соответствующих операций, как успешных, так и ошибочных.

В JFS2 существуют функции, позволяющие заранее выделить ресурсы для метаданных в файловой системе и управлять ими непосредственно. Вызов соответствующих функций в режимах MM_ALLOC или MM_RECORD генерирует событие записи DMAPI для указанных значений смещения и длины.

Если маска для функций **dm_get_bulkall**, **dm_get_bulkattr**, **dm_get_dirattrs** и **dm_get_fileattr** не задана (то есть равна нулю), то JFS2 возвратит все поля структуры **dm_stat**. В противном случае будут возвращены значения полей, указанных в маске, а значения других полей будут не определены.

В JFS2 параметр **respbufp** функции **dm_respond_event** не используется. Если он задан, то при выходе из функции содержимое буфера будет не определено.

Поскольку в JFS2 dm_ctime и dm_dtime переопределяются (то есть DM_CONFIG_DTIME_OVERLOAD равно true), параметр **setdtime** функции **dm_set_dmattr** игнорируется.

При отображении в память файла (при вызове функции **mmap(2)**) приложение управления данными должно сделать резидентными все нерезидентные части файла. JFS2 известит приложение об отображении с помощью события чтения или записи в соответствии с режимом отображения и отображаемой области.

Активация DMAPI в JFS2

Для того чтобы активировать DMAPI в JFS2, выполните следующую команду:

```
chfs -a managed=yes mountpoint
```

Если файловая система смонтирована, и запускается команда **chfs**, то при заданном параметре *managed* должно быть запущено приложение с поддержкой DMAPI, которое принимает и обрабатывает события монтирования. От того, как приложение обрабатывает события монтирования, зависит успешное выполнение команды **chfs**.

Для того чтобы деактивировать DMAPI в JFS2, выполните следующую команду:

```
chfs -a managed=no mountpoint
```

Если файловая система смонтирована, и запускается команда **chfs**, то при заданном параметре *managed* должно быть запущено приложение с поддержкой DMAPI, которое принимает и обрабатывает события, предшествующие размонтированию. Успешное выполнение команды **chfs** зависит от ответа приложения на события, предшествующие размонтированию.

Работа с DMAPI в файловых системах JFS2 с шифрованием

При выполнении операций невидимого ввода-вывода над зашифрованными файлами в файловой системе JFS2 с шифрованием, действуют те же ограничения по смещению и выравниванию по длине, что и при выполнении операций над файлом в режиме ПРЯМОГО ввода-вывода. В частности, смещение и длина ввода-вывода должны быть упорядочены по блокам в соответствии с размером блока, установленным в файловой системе. Размер зашифрованных данных всегда кратен размеру блока файловой системы, даже если размер расшифрованного файла не кратен размеру блока; если размер файла не соответствует размеру блока, такой файл содержит зашифрованные данные сверх размера файла.

Примечание: Функция **stat** и интерфейсы DMAPI, например, функция **dm_get_fileattr**, возвращают размер файла с открытым текстом (расшифрованного), тогда как функция **statx** возвращает размер зашифрованных данных, упорядоченных по блокам, при указании значения **STX_EFSRAW** для параметра *command*.

Для успешного выполнения операции функции **dm_read_invis** и **dm_write_invis** должны отвечать следующим требованиям:

dm_read_invis
: Для зашифрованных файлов, параметры *off* и *len* должны соответствовать размеру блока файловой системы, иначе операция завершится сбоем и будет выдан код ошибки **EINVAL**.

dm_write_invis
: Для зашифрованных файлов, параметры *off* и *len* должны соответствовать размеру блока файловой системы, и операция не должна предпринимать попыток расширить данный файл. В противном случае произойдет сбой и будет выдан код ошибки **EINVAL**.

Работа с DMAPI в рабочих разделах AIX

Для того чтобы в рабочем разделе WPAR можно было запускать приложения DMAPI, в набор привилегий необходимо добавить **PV_FS_DMAPI** и присвоить их всем процессам, выполняющимся в WPAR. Набор привилегий можно присвоить WPAR при создании или изменить уже присвоенный набор позднее.

Примеры

```
mkwpar -S privs+=PV_FS_DMAPI -n имя-wpar
chwpar -S privs+=PV_FS_DMAPI имя-wpar
```

По умолчанию, выполняться в WPAR могут только процессы root. В системах с отключенным root или в Trusted AIX, где пользователь root отключен по умолчанию, другим процессам можно разрешить выполнение в WPAR с помощью таблицы **privcmds** в глобальной системе или в WPAR. Дополнительная информация приведена в разделе Привилегии RBAC.

Программирование транзакционной памяти AIX

Транзакционная память - это механизм синхронизации общей памяти, который позволяет нитям процесса выполнять операции сохранения, которые воспринимаются другими нитями процесса или приложениями как атомарные.

Обзор

Транзакционная память - это конструкция, которая позволяет выполнять критические разделы кода, основанных на блокировках, без получения блокировки. Первый процессор, реализующий программирование транзакционной памяти - это IBM POWER 8.

Используйте механизм транзакционной памяти в следующих сценариях:

- Оптимистическое выполнение приложений, основанных на блокировках. Транзакционная память поддерживает спекулятивное выполнение (выполнение "по предположению") критических разделов кода без получения блокировки. Этот метод предоставляет приложениям преимущества блокировки с высокой степенью детализации путем использования текущих блокировок, которые не настроены для производительности.
- Программирование транзакций на языках высокого уровня. Программная модель транзакций - это развивающийся общеотраслевой стандарт, который предлагает повышение производительности относительно программ доступа к общей памяти на основе блокировки.
- Использование контрольной точки/отката – Транзакционная память используется в качестве контрольной точки для восстановления архитектурного состояния. Этот метод позволяет выполнять спекулятивные оптимизации при компиляции во время динамической оптимизации кода или генерации и имитации контрольных точек.

Для использования механизма транзакционной памяти нить процесса помечает начало и конец последовательности обращений к памяти или транзакции с помощью команд `tbegin.` и `tend.` . Команда `tbegin.` инициирует выполнение транзакции, во время которого операции загрузки и сохранения выполняются как атомарные. Команда `tend.` завершает выполнение транзакции.

При преждевременной остановке транзакции выполняется откат обновлений памяти, сделанных после выполнения команды `tbegin.`. Соответственно, для содержимого подмножества регистров также выполняется откат в состояние, которое было до выполнения команды `tbegin.`. При преждевременной остановке транзакции запускается обработчик программных сбоев. Сбой может быть устойчивым или неустойчивым. Обработчик сбоев может повторить транзакцию или выбрать использование другой блокирующей логической структуры или другого логического пути, которые зависят от характера сбоя.

Операционная система AIX поддерживает использование транзакционной памяти, включая обработку управления состоянием транзакционной памяти между переключениями контекста и прерываниями.

Состояние контрольной точки

При инициации транзакции сохраняется набор регистров, представляющих состояние контрольной точки процессора. В случае сбоя транзакции восстанавливается состояние регистров, существовавшее перед запуском транзакции. Состояние контрольной точки процессора называется также предтранзакционным. Состояние контрольной точки включает перезаписываемые регистры задачного режима, за исключением регистров `CR0`, `FXCC`, `EBBHR`, `EBBRR`, `BESCR`, регистры системного монитора и SPR транзакционной памяти.

Примечание: К состоянию контрольной точки нельзя получить доступ непосредственно через состояние супервизора или состояние задачи.
После выполнения новой команды `treclaim.` состояние контрольной точки копируется в соответствующие регистры . Этот процесс позволяет привилегированному коду сохранять и изменять значения. После выполнения новой команды `trechkpt.` состояние контрольной точки копируется обратно в предполагаемые регистры из соответствующих пользовательских регистров .

К регистрам состояния машины процессора добавляются следующие SPR транзакционной памяти:

Имя	Название	Описание	Привилегированный mtspr	Привилегированный mfspr	Размер (бит)	SPR
FSCR	Facility Status and Control Register (регистр состояния устройств и управления)	Управляет доступными устройствами в состоянии задачи и указывает причину прерывания из-за недоступности устройства.	да	да	64	153
TEXASR	Transaction Exception And Summary Register (регистр исключительной ситуации транзакции и сводной информации)	Содержит информацию об уровне транзакции и сводную информацию, которая используется обработчиками сбоев транзакций. Биты 0:31 содержат причину сбоя.	нет	нет	64	130
TFHAR	Transaction Failure Handler Address Register (адресный регистр обработчика сбоев транзакций)	Записывает EA обработчика программных сбоев. Для команды `tbegin.`, которая инициировала транзакцию, для регистра TFHAR всегда задается NIA.	нет	нет	64	128
TFIAR	Transaction Failure Instruction Address Register (регистр адреса команды, вызвавшей сбой транзакции)	Если это возможно, содержит точное значение EA команды, которая вызвала сбой. Точность регистра TFIAR указана в поле Exact (бит 37) регистра TEXASR.	нет	нет	64	129
TEXASRU	Transaction Exception And Summary Register (регистр исключительной ситуации транзакции и сводной информации, верхняя половина)	Верхняя половина регистра TEXASR.	нет	нет	32	131

Новый регистр TEXASR содержит информацию, относящуюся к состоянию транзакции и причине ее сбоя. Поля, включаемые в регистр TEXASR, описаны в следующей таблице:

Имя	Поле	Смысл значения	Биты
TEXASR	Код сбоя (примечание: бит 7 называется полем Устойчивый сбой)	Коды сбоев транзакций	0:7
	Запрещено	0b1 - команда типа доступа не разрешена	8
	Переполнение при вложении	0b1 - превышен максимальный уровень транзакции.	9
	Переполнение потребности	0b1 - превышен лимит отслеживания для обращений к транзакционной памяти.	10
	Самопроизвольный конфликт	0b1 - в состоянии приостановки произошел самопроизвольный конфликт.	11
	Нетранзакционный конфликт	0b1 - возник конфликт с доступом другого процессора в нетранзакционном режиме.	12
	Конфликт транзакций	0b1 - Возник конфликт с другой транзакцией.	13
	Конфликт при аннулировании преобразования	0b1 - произошел конфликт при аннулировании TLB.	14
	Конкретная реализация	0b1 - зависящее от реализации условие привело к сбою транзакции.	15
	Конфликт при вызове команды	0b1 - вызов команды нитью или другой нитью, выполнявшийся из блока, который ранее был записан в режиме транзакции.	16
	Зарезервировано для будущих вариантов сбоев		17:30
	Прекращение выполнения	0b1 – преждевременное прекращение выполнения было вызвано выполнением команды транзакционной памяти.	31
	Приостановка	0b1 – В состоянии приостановки был зарегистрирован сбой.	32
	Зарезервировано		33
	Привилегия	Во время регистрации сбоя нить находилась в привилегированном состоянии (`[MSRHV||PR]`).	34:35
	Сводная информация о сбое (FS)	0b1 - сбой был обнаружен и записан.	36
	Точность TFIAR	0b0 - поле TFIAR содержит приближенное значение. 0b1 - поле TFIAR содержит точное значение.	37
	ROT	Значение равно 0b0 для отличной от ROT `tbegin.` . При инициации транзакции ROT значение равно 0b1.	38
	Зарезервировано		39:51
	Уровень транзакции (TL)	Уровень транзакции (уровень вложенности + 1) для активной транзакции имеет следующие значения: • 0, если самая последняя транзакция успешно выполнена. • Уровень транзакции, на котором не была выполнена самая последняя транзакция, если транзакция не была выполнена успешно. **Примечание:** Значение 1 соответствует *внешней транзакции*. Значение, превышающее 1, соответствует *вложенной транзакции*.	52:63

Notes:

- В случае регистрации сбоя транзакции задается в точности 1 бит из 8-31 битов регистра TEXASR. Единственный заданный бит указывает, что сбой вызван конкретной командой или событием.
- Транзакция ROT (Rollback Only Transaction) - это последовательность команд, которые выполняются или не выполняются единым блоком. Эта конструкция позволяет с минимальными затратами выполнять большой объем команд "по предположению". В отличие от обычной транзакции, ROT не является полностью атомарной и не обладает свойствами синхронизации и сериализации, присущими обычной транзакции. Поэтому транзакции ROT не следует использовать для обработки общих данных.

Обработчик программных сбоев

В случае сбоя транзакции аппаратное обеспечение системы передает управление обработчику сбоев, связанному с самой внешней транзакцией. При сбое транзакции управление передается команде, которая следует за командой tbegin., а в CR0 задается или

```
0b101 || 0
```

, или

```
0b010 || 0
```

Следовательно, за командой tbegin. должна следовать команда передачи управления, основанная на бите 2 регистра CR0. Например, после выполнения команды tbegin. может следовать команда передачи управления beq, основанная на бите 2 регистра CR0. Управление должно передаваться разделу кода, который обрабатывает сбои транзакций. Если в начале транзакции успешно выполняется команда tbegin., то в регистре CR0 устанавливается или

```
0b000 || 0, или 0b010 || 0
```

Примечание: На причину сбоя указывают биты 0:31 TEXASR. Поле кода сбоя (FC) в битах 0-7 используется для следующих сценариев:

- Привилегированный код в режиме супервизора или гипервизора вызывает сбой при использовании команды treclaim. .
- Код в режиме задачи вызывает сбой при использовании формы tabort. .

Значение l в седьмом бите регистра TEXASR указывает, что сбой является устойчивым, и повторные попытки выполнить транзакцию будут неуспешными. Коды сбоев, зарезервированные операционной системой AIX, указывают причину сбоя, которая определена в файле /usr/include/sys/machine.h..

Пример транзакции

Следующий пример кода на ассемблере иллюстрирует простую транзакцию, которая записывает значение из GPR 5 по адресу в GPR 4, который считается общим для нескольких нитей выполнения. В случае невыполнения транзакции из-за устойчивого сбоя программа выполняется по альтернативному пути с меткой lock_based_update. Код для альтернативного пути не показан.

```
trans_entry:
  tbegin                          # Запустить транзакцию
  beq           failure_hdlr      # Обработать сбой транзакции
  stw           r5, 0(r4)         # Записать в память, указанную r4.
  tend.                           # Завершить транзакцию
  b             trans_exit
failure_hdlr:                     # Обработать сбои транзакции:
  mfspr         r4, TEXASRU       # Прочитать верхнюю половину TEXASR
  andis.        r5, r4, 0x0100    # Является ли сбой устойчивым?
  bne           lock_based_update # Если сбой является устойчивым, получить
                                  # блокировку, затем выполнить запись.
  b             trans_entry       # Если это самоустраняющийся отказ, повторить транзакцию.
```

```
lock_based_update:

trans_exit:
```

Динамическая проверка поддержки транзакционной памяти

Для того чтобы определить, поддерживает ли система категорию транзакционной памяти POWER ISA, в программе можно использовать функцию `getsystemcfg` для чтения системной переменной `SC_TM_VER`. В файле `/usr/include/sys/systemcfg.h` предоставляется макрос `__power_tm()`, предназначенный для определения функции транзакционной памяти внутри программы. Этот макрос полезен для программного обеспечения, которое условно использует функцию транзакционной памяти при ее наличии, а при ее отсутствии использует функционально эквивалентные пути кода на основе блокировки.

Структура расширенного контекста

В более ранних версиях операционной системы AIX была введена поддержка структур расширенного контекста для поддержки состояния векторов и пользовательских ключей. Существующая поддержка структур расширенного контекста расширена для поддержки состояния системы, которое требуется транзакционной памяти.

Расширенный контекст выделяется и закрепляется для каждой нити транзакционного процесса при первом использовании ею транзакционной памяти. Если область расширенного контекста невозможно выделить и закрепить, то процесс получает сигнал `SIGSEGV`, который приводит к завершению процесса.

Информация о контексте системы включается в структуру `sigcontext`, которая предоставляется обработчикам сигналов. При возврате из обработчика сигнала активируется контекст системы, представленный в структуре `sigcontext`. Фактически структура `sigcontext` представляет собой подмножество структуры `ucontext` большего размера. Эти две структуры идентичные; различие состоит только в результате `sizeof(struct sigcontext)`. Когда операционная система AIX компонует контекст сигналов, передаваемый обработчику сигналов, в стеке обработчика сигналов компонуется структура `ucontext`. Контекст системы, входящий в состав контекста сигналов, должен содержать все сведения о динамическом или статическом состоянии системы для случайно прерванного контекста. Структура `ucontext` содержит индикатор, указывающий, доступна ли информация о расширенном контексте.

Поле `__extctx` структуры `ucontext` - это адрес структуры расширенного контекста, которая определена в файле `/usr/include/sys/context.h`. Поле `__extctx_magic` структуры `ucontext` указывает, допустима ли информация о расширенном контексте, когда значение поля `__extctx_magic` равно `__EXTCTX_MAGIC`. Дополнительное состояние системы для нити, которая использует функцию транзакционной памяти, восстанавливается и сохраняется как элемент расширения контекста в структуре `ucontext` в рамках доставки и возврата сигнала.

Если приложение явным образом разрешает использование транзакционной памяти, то оно выбирает структуру `ucontext` увеличенного размера, в которой уже есть память для поля `__extctx`, которое включается компилятором путем неявного определения `__EXTABI__`. Расширенная структура `ucontext` может также выбираться с помощью явного определения `__AIXEXTABI`.

В состоянии выполнения транзакции и в состоянии приостановки транзакции не поддерживаются функции libc `getcontext()`, `setcontext()`, `makecontext()` и `swapcontext()`. Вызов функций `getcontext()`, `setcontext()`, `makecontext()` внутри транзакции приводит к устойчивому сбою транзакции типа `TM_LIBC`, который определен в файле `/usr/include/sys/machine.h`.

Вызов функции `swapcontext()` внутри транзакции приводит к следующему поведению:

- Вызов функции `swapcontext()` в состоянии выполняющейся транзакции приводит к устойчивому сбою транзакции типа `TM_LIBC`.

- Если функция swapcontext() вызывается в состоянии приостановки, то неудачное выполнение транзакции предопределено; подкачивается указанная структура ucontext, и возобновляется выполнение программы указанной структурой ucontext. Итоговое состояние и последующее поведение после возврата из функции swapcontext() не определено.

Если функции getcontext(), setcontext() и swapcontext() вызываются не в транзакционном состоянии, то они не извлекают в структуру ucontext и не восстанавливают из нее никакой расширенный контекст транзакционной памяти (структура указывается параметрами ucp или oucp). Если функции setcontext() или swapcontext() вызываются в присутствии расширенного контекста транзакционной памяти, то никакая ошибка не выдается.

Подробная информация о расширенном контексте содержится в заголовочном файле /usr/include/sys/context.h.

Доставка сигналов

Асинхронные сигналы, получаемые приложением в режиме выполнения транзакции, доставляются нетранзакционным способом. В состоянии, когда выполняется транзакция, доставка синхронных сигналов не разрешена, так как она приводит к устойчивому сбою транзакции типа TM_SYNC_SIGNAL, определенному в файле /usr/include/sys/machine.h.

Прерывания выравнивания и программные прерывания

В состоянии выполняющейся транзакции недопустимая операция или операция, требующая эмуляции, вызывает прерывания выравнивания или программные прерывания, что приводит к устойчивому сбою транзакции типа TM_ALIGN_INT или TM_INV_OP, который определен в файле /usr/include/sys/machine.h. В состоянии приостановленной транзакции прерывания выравнивания и программные прерывания обрабатываются обычным образом с использованием неспекулятивной семантики.

Системные вызовы

Предполагается, что транзакция не содержит системных вызовов. Системные вызовы в транзакции поддерживаются только в том случае, если транзакция приостановлена с помощью команды tsuspend. .

Если процесс или нить находятся в состоянии транзакции, и транзакция не приостановлена, то системный вызов в этом состоянии не выполняется ядром AIX и возникает устойчивый сбой транзакции. При возникновении этой ошибки поле FC регистра TEXASR содержит код сбоя TM_ILL_SC, который определен в файле /usr/sys/include/machine.h.

Предполагается, что сохраняются любые операции, выполняемые в транзакции, когда она явно приостанавливается разработчиком приложения. Любые операции, которые выполняются системным вызовом, сделанным в состоянии приостановки, не откатываются, даже если происходит сбой транзакции.

Операционная система AIX не поддерживает системные вызовы в транзакционном состоянии, поскольку не существует способа отката каких-либо операций, включая операции ввода-вывода, выполняемых AIX при системном вызове.

Функции setjmp() и longjmp()

В транзакционном состоянии и в состоянии приостановки не поддерживаются функции libc setjmp() и longjmp() из-за эффектов задания буфера перехода (jump buffer) и перехода обратно в буфер. Рассмотрим следующие сценарии

1. Если функция setjmp() вызывается внутри транзакции, а соответствующая функция longjmp() - после завершения транзакции, то выполняется переход в спекулятивное состояние, которое теперь недопустимо.

2. Если функция setjmp() вызывается перед транзакцией, соответствующая функция longjmp() переходит в это состояние до начала транзакции, независимо от результата выполнения транзакции: успешного завершения, сбоя или аварийного завершения.
3. Если функция setjmp() вызывается внутри транзакции, а затем выполнение транзакции прекращается, то обновления, внесенные в буфер перехода функцией setjmp(), будут считаться не имевшими места.

Вызов функции setjmp() внутри транзакции приводит к устойчивому сбою транзакции типа TM_LIBC, или типа TM_ILL_SC, который определен в файле /usr/include/sys/machine.h.

Вызов функции longjmp() внутри транзакции приводит к следующему поведению:

- Вызов функции longjmp() в состоянии выполняющейся транзакции приводит к устойчивому сбою транзакции типа TM_LIBC или типа TM_ILL_SC, который определен в файле /usr/include/sys/machine.h.
- Если функция longjmp() вызывается в состоянии приостановки, то неудачное выполнение транзакции предопределено; указанный буфер восстанавливается и выполнение программы возвращается в соответствующую функцию setjmp(). Итоговое состояние и последующее поведение после возврата к setjmp() из функции longjmp() не определено.

Компиляторы

Компилятор операционной системы AIX, который поддерживает транзакционную память, должен соответствовать ABI AIX. Если включена транзакционная память, то компилятор C или C++ должен предопределить __EXTABI__. За более подробной информацией обратитесь к документации по компилятору.

Ассемблер

Компилятор ассемблера операционной системы AIX, расположенный в каталоге /usr/ccs/bin/as, поддерживает дополнительный набор инструкций, определенный для транзакционной памяти в POWER ISA и реализуемый процессором POWER8. Для того чтобы включить сборку команд транзакционной памяти, можно использовать режим сборки **–m pwr8** или псевдооперацию **.machine pwr8** внутри файла исходного кода. За дополнительной информацией обратитесь к справочнику по языку ассемблера.

Отладчик

Отладчик /usr/ccs/bin/dbx поддерживает отладку программ транзакционной памяти на машинном уровне. Эта поддержка включает возможность дизассемблирования новых команд транзакционной памяти и просмотра ее регистров специального назначения (SPR): TEXASR, TEXASRU, TFIAR и TFHAR.

Задание точки прерывания внутри транзакции приводит к безусловному прекращению выполнения транзакции при обнаружении точки прерывания. Таким образом, предлагаемый подход заключается в следующем: запуск отладчика для транзакции, которая не выполняется; задание точки прерывания в обработчике сбоев транзакции; просмотр регистров TEXASR и TFIAR в точке прерывания для определения причины и расположения сбоя.

В dbx для просмотра регистров TEXASR, TFIAR и TFHAR можно использовать команду print с параметром $texasr, $tfiar или $tfhar. Строку кода, связанную с адресом, найденным в регистрах TFIAR и TFHAR, можно просмотреть с помощью команды list, например:

```
(dbx) list at $tfiar
```

Для просмотра и интерпретации содержимого регистра TEXASR применяется команда dbx tm_status. Эта команда позволяет определить характер сбоя транзакции.

Для подключения отладчиков других фирм предусмотрена новая операция PTT_READ_TM ptrace для считывания состояния транзакционной памяти нити. За информацией обратитесь к документации по ptrace.

Поддержка трассировки

Трассировщик AIX был расширен и теперь включает набор событий трассировки операций транзакционной памяти, которые выполняются операционной системой AIX, в том числе приоритетное переключение процессов и другие операции, вызывающие сбой транзакции. Для просмотра событий трассировки, относящихся к транзакционной памяти, укажите в качестве входного параметра команды **trace** и **trcrpt** идентификатор события трассировки 675.

Файлы дампа

Операционная система AIX поддерживает включение состояния системы транзакционной памяти в файл дампа для использующих эту память процессов или нитей. Если процесс или нить используют или использовали транзакционную память, то в образ оперативной памяти включается состояние системы транзакционной памяти для такой нити.

Примечание: Состояние транзакционной памяти поддерживается только в текущих форматах файлов дампа для операционной системы AIX. Для чтения файла дампа с поддержкой транзакционной памяти и просмотра состояния системы транзакционной памяти применяется команда **dbx**.

Библиотека нитей AIX

Для приложений, которые используют нити M:N, транзакционная память не поддерживается. В среде, где несколько нитей совместно используют одну нить ядра, поведение транзакционных нитей может стать неопределенным. Применение транзакционной памяти приложением, которое использует нити M:N, может привести к устойчивому сбою транзакции с кодом сбоя `TM_PTH_PREEMPTED`, задаваемым в регистре `TEXASR`.

Примечания

Данная информация была разработана для продуктов и услуг, предлагаемых на территории США.

Компания IBM может не предоставлять в других странах продукты и услуги, обсуждаемые в данном документе. Информацию о продуктах и услугах, распространяемых в вашей стране, вы можете получить в местном представительстве IBM. Ссылки на продукты, программы или услуги IBM не означают, что можно использовать только указанные продукты, программы или услуги IBM. Вместо них можно использовать любые другие функционально эквивалентные продукты, программы или услуги, не нарушающие прав IBM на интеллектуальную собственность. Однако ответственность за проверку действия любых продуктов, программ и услуг других компаний лежит на пользователе.

Компания IBM может обладать заявками на патенты или патентами на предметы обсуждения в данном документе. Обладание данным документом не предоставляет лицензии на эти патенты. Запросы на получение лицензии можно отправлять в письменном виде по адресу:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

За получением лицензий, имеющих отношение к двухбайтовому набору символов (DBCS), обращайтесь в местное отделение компании IBM по интеллектуальной собственности или направьте запрос в письменной форме по следующему адресу:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

КОМПАНИЯ IBM ПРЕДОСТАВЛЯЕТ НАСТОЯЩУЮ ПУБЛИКАЦИЮ НА УСЛОВИЯХ "КАК ЕСТЬ", БЕЗ КАКИХ-ЛИБО ЯВНЫХ ИЛИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ЭТИМ, НЕЯВНЫЕ ГАРАНТИИ СОБЛЮДЕНИЯ ПРАВ, КОММЕРЧЕСКОЙ ЦЕННОСТИ И ПРИГОДНОСТИ ДЛЯ КАКОЙ-ЛИБО ЦЕЛИ. В некоторых юрисдикциях освобождение от явных и подразумеваемых гарантий запрещено в некоторых сделках, поэтому это заявление может к вам не относиться.

Эта информация может содержать технические неточности или типографические ошибки. В информацию периодически вносятся изменения, которые будут учтены во всех последующих изданиях этой книги. IBM может вносить обновления или изменения в этот документ без предварительного уведомления.

Любые ссылки на веб-сайты других компаний приведены в данной публикации исключительно для удобства пользователей и не должны рассматриваться как рекомендация этих веб-сайтов. Материалы, размещенные на этих веб-сайтах, не являются частью информации по данному продукту IBM, и ответственность за применение этих материалов лежит на пользователе.

IBM может использовать и распространять предоставленную вами информацию любым способом без каких-либо обязательств перед вами.

Лицам, обладающим лицензией на данную программу и желающим получить информацию о ней с целью: (i) настройки обмена данными между независимо разработанными программами и другими программами (включая данную) и (ii) использования информации, полученной в результате обмена, этими программами, следует обращаться по адресу:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Такая информация может быть предоставлена на определенных условиях, а в некоторых случаях - и за дополнительную плату.

Описанная в этом документе лицензионная программа и все связанные с ней лицензионные материалы предоставляются IBM в соответствии с условиями Соглашения с заказчиком IBM, Международного соглашения о лицензии на программу IBM или любого другого эквивалентного соглашения.

Данные о производительности и примеры клиентов приведены исключительно в иллюстративных целях. Фактические результаты производительности зависят от конкретных конфигураций и рабочих сред.

Информация о продуктах других компаний была получена от поставщиков этих продуктов, их опубликованных материалов или других общедоступных источников. Компания IBM не проверяла эти продукты и не может подтвердить правильность их работы, совместимость или другие заявленные характеристики продуктов других компаний. По вопросам о возможностях продуктов других компаний следует обращаться к поставщикам этих продуктов.

Заявления относительно будущих намерений IBM могут быть изменены или отозваны без дополнительного уведомления и отражают только текущие цели и задачи.

Все указанные цены IBM являются рекомендуемыми розничными ценами IBM на данный момент и могут быть изменены без предварительного уведомления. Цены дилеров могут быть другими.

Данная информация предназначена только для планирования. Она может быть изменена до выпуска описанных в данном документе продуктов.

Настоящая документация содержит примеры данных и отчетов, применяемых в повседневной деятельности компаний. Для большего сходства с реальностью примеры содержат имена людей, названия компаний, товарных знаков и продуктов. Все эти имена и названия вымышленные. Любые совпадения с реально существующими физическими или юридическими лицами совершенно случайны.

Лицензия на авторские права:

Настоящая документация содержит примеры исходного кода программ, иллюстрирующие приемы программирования в различных операционных системах. Вы имеете право копировать, изменять и распространять эти примеры программ в любой форме без уплаты вознаграждения фирме IBM в целях разработки, применения, сбыта или распространения прикладных программ, соответствующих интерфейсу прикладных программ операционной системы, для которой предназначены эти примеры. Эти примеры не были тщательно и всесторонне протестированы. В связи с этим IBM не может гарантировать их надежность, удобство обслуживания и отсутствие ошибок. Примеры программ предоставляются "КАК ЕСТЬ", без каких-либо гарантий. IBM не несет ответственности за ущерб, который может возникнуть в результате использования эти образцов программ.

Во все копии или фрагменты этих примеров программ, а также программы созданные на их основе, следует добавлять следующее замечание об авторских правах:

© (название вашей компании) (год).

Некоторые фрагменты исходного кода получены из примеров программ фирмы IBM Corp.

Замечания о правилах работы с личными данными

Продукты IBM Software, включая решения программного обеспечения как услуг, (“Предложения программного обеспечения”) могут использовать cookie или другие технологии для сбора информации об использовании продукта в целях усовершенствования пользовательского интерфейса, для приспособления взаимодействий к конечному пользователю или для других целей. Во многих случаях Предложениями программного обеспечения собирается информация, в которой невозможно опознать персональные данные. Некоторые из наших Предложений программного обеспечения могут позволить вам собирать опознаваемую персональную информацию. Если это Предложение программного обеспечения использует cookie для сбора опознаваемой персональной информации, то специфическая информация об этом использовании cookie в предложении приведена далее.

Это Предложение программного обеспечения не использует cookie или другие технологии для сбора опознаваемой персональной информации.

Если конфигурации, развернутые для этого Предложения программного обеспечения предоставляют вам как клиенту возможность собирать опознаваемую персональную информацию о конечных пользователях посредством cookie и других технологий, вы должны самостоятельно проконсультироваться с юристом о всех законах, применимых к такому сбору данных, включая требования к уведомлению и согласию.

Более подробная информация об использовании различных технологий, включая cookie, для этих целей, приведена в Политике конфиденциальности IBM (http://www.ibm.com/privacy) и Заявлении IBM о конфиденциальности в Интернет (http://www.ibm.com/privacy/details), а также в разделах “Cookies, Web Beacons and Other Technologies” и “IBM Software Products and Software-as-a-Service Privacy Statement” на странице http://www.ibm.com/software/info/product-privacy.

Товарные знаки

IBM, эмблема IBM и ibm.com являются товарными знаками или зарегистрированными товарными знаками International Business Machines Corp. во всем мире. Названия других продуктов и услуг могут быть товарными знаками IBM и других компаний. Текущий список товарных знаков IBM опубликован на веб-странице Copyright and trademark information по следующему адресу: www.ibm.com/legal/copytrade.shtml.

Intel, эмблема Intel, Intel Inside, эмблема Intel Inside, Intel Centrino, эмблема Intel Centrino, Celeron, Intel Xeon, Intel SpeedStep, Itanium и Pentium являются товарными знаками или зарегистрированными товарными знаками Intel Corporation и ее дочерних фирм в США и/или других странах.

Java и все основанные на Java названия и эмблемы являются товарными знаками или зарегистрированными товарными знаками Oracle и/или дочерних компаний.

UNIX - зарегистрированный товарный знак The Open Group в США и других странах.

Индекс

Спец. символы

Числа

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

A

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ч

Э

Напечатано в Дании