SCO OpenServer™ Operating System User's Guide

SCO OpenServer™

SCO OpenServer™
Operating System User's Guide

Document Version: 5.0
1 May 1995

About this book   1

Chapter 1
Using SCO Shell   11

Chapter 2

SCO Shell accessories 49

Chapter 3

Working with files and directories 79

Chapter 4

Editing files   131

Chapter 10
Configuring and working with the shells   221

Chapter 11

Automating frequent tasks 245

Chapter 12

Regular expressions 315

Chapter 13

Using awk 323

Chapter 14

Manipulating text with sed 371

Appendix A

An overview of the system 393

About this book

This User's Guide contains an introduction to using the SCO Operating System. It explains how to accomplish routine tasks, and provides more detailed information than the *Operating System Tutorial*. You will find the information you need more quickly if you are familiar with:

- "How this book is organized" (this page)
- "Related documentation" (page 3)
- "Typographical conventions" (page 6)

Although we try to present information in the most useful way, you are the ultimate judge of how well we succeed. Please let us know how we can improve this book (page 7).

This book is clearly too short to be a full reference to the system. There are in excess of five hundred commands, and two thousand files in a basic operating system; however, for most activities you only need to be familiar with a handful of them.

How this book is organized

This book, which is divided into three sections, is designed to lead you through the workings of the SCO Operating System, from using its office automation facilities to set up your calendar, to writing simple shell scripts that perform tasks for you.

Office automation

The SCO OpenServer™ system contains a number of powerful office automation tools that allow you to manage your work. Chapter 1, "Using SCO Shell" (page 11) contains a guide to using the SCO office automation tools. Chapter 2, "SCO Shell accessories" (page 49) explains how to use the Calendar and Calculator accessories provided with the SCO Shell.

Working at the shell prompt

The shells are powerful programs that you can use to issue commands directly to the SCO OpenServer system. The chapters in this section, beginning with Chapter 3, "Working with files and directories" (page 79), introduce you to the shells and explain how to use them to run a variety of programs which are present on your system. This section also explains the basic concepts of file storage and manipulation, and how to manage your work environment effectively.

Shell programming

The three available shells provide a powerful but simple programming language that you can use to automate complex tasks, write your own commands, and connect other programs together to perform a sequence of operations. This section contains the following chapters:

- Chapter 10, "Configuring and working with the shells" (page 221), — an explanation of the different shells you may be working in, and their special features

- Chapter 11, "Automating frequent tasks" (page 245) — examples and explanations of how to write simple scripts

- Chapter 12, "Regular expressions" (page 315) — the extensive pattern matching facilities that these tools use to identify data

- Chapter 13, "Using awk" (page 323) — a powerful but complex tool provided for manipulating and reporting on textual data)

- Chapter 14, "Manipulating text with sed" (page 371) — a stream editor, used for rapidly making changes to large files

Appendices

The following appendices are provided:

- Appendix A, "An overview of the system" (page 393) contains useful background material for the main text. It explains the basic history and design philosophy of the SCO OpenServer system; what its components are, what they do, and how they all work together to provide your work environment.

- Appendix B, "vi commands" (page 409) provides a concise listing of the commands recognized by the **vi** text editor.

- Appendix C, "DOS command equivalents" (page 415) provides a table showing common MS-DOS® commands and their SCO OpenServer system equivalents.

- Appendix D, "Sample shell startup files" (page 419) contains some sample listings and explanations of the standard user shell startup files.

- Appendix E, "Further reading" (page 427) contains references to sources of further information that lie beyond the scope of this book.

Related documentation

SCO OpenServer systems include comprehensive documentation. Depending on which SCO OpenServer system you have, the following books are available in online and/or printed form. Access online books by double-clicking on the Desktop **Help** icon. Additional printed versions of the books are also available. The Desktop and most SCO OpenServer programs and utilities are linked to extensive context-sensitive help, which in turn is linked to relevant sections in the online versions of the following books. See "Getting help" in the *SCO OpenServer Handbook*.

> **NOTE** When you upgrade or supplement your SCO OpenServer software, you might also install online documentation that is more current than the printed books that came with the original system. In particular, the new information provided online with our regular Advanced Hardware Supplements (AHS) supersedes and frequently obsoletes the material in the printed version of this book. For the most up-to-date information, check the online documentation.

Release Notes
> contain important late-breaking information about installation, hardware requirements, and known limitations. The *Release Notes* also highlight the new features added for this release.

Operating System Tutorial
> provides a basic introduction to the SCO OpenServer operating system. This book can also be used as a refresher course or a quick-reference guide. Each chapter is a self-contained lesson designed to give hands-on experience using the SCO OpenServer operating system.

Graphical Environment Help
> describes how to use Calendar, Edit, the Desktop, Help, Mail, Paint, the SCO Panner window manager, and the UNIX command-line window.

Operating System User's Reference
> contains the manual pages for user-accessible operating system commands and utilities (section C).

SCO OpenServer Handbook
> provides the information needed to get your SCO OpenServer system up and running, including installation and configuration instructions, and introductions to the Desktop, online documentation, system administration, and troubleshooting.

Mail and Messaging Guide
> describes how to configure and administer your mail system. Topics include **sendmail**, MMDF, SCO Shell **Mail**, **mailx**, and the Post Office Protocol (POP) server.

Guide to Gateways for LAN Servers
> describes how to set up SCO® Gateway for NetWare® and LAN Manager Client software on an SCO OpenServer system to access printers, filesystems, and other services provided by servers running Novell® NetWare® and by servers running LAN Manager over DOS, OS/2®, or UNIX® systems.

PC-Interface Guide
> describes how to set up PC-Interface™ software on an SCO OpenServer system to provide print, file, and terminal emulation services to computers running PC-Interface client software under DOS or Microsoft® Windows™.

Graphical Environment Guide
> describes how to customize and administer the Graphical Environment, including the X Window System™ server, the SCO® Panner™ window manager, the Desktop, and other X clients.

Graphical Environment Reference
> contains the manual pages for the X server (section XS), the SCO Panner window manager, Desktop, and X clients from SCO and MIT (section XC).

Networking Guide
> provides information on configuring and administering TCP/IP, NFS®, and IPX/SPX™ software to provide networked and distributed functionality, including system and network management, applications support, and file, name, and time services.

Networking Reference
> contains the command, file, protocol, and utility manual pages for the IPX/SPX (section PADM), NFS (sections NADM, NC, and NF), and TCP/IP (sections ADMN, ADMP, SFF, and TC) networking software.

System Administration Guide
> describes configuration and maintenance of the base operating system, including account, filesystem, printer, backup, security, UUCP, and virtual disk management.

Operating System Administrator's Reference
> contains the manual pages for system administration commands and utilities (section ADM), system file formats (section F), hardware-specific information (section HW), miscellaneous commands (section M), and SCO Visual Tcl™ commands (section TCL).

Performance Guide
> describes performance tuning for uniprocessor, multiprocessor, and networked systems, including those with TCP/IP, NFS, and X clients. This book discusses how the various subsystems function, possible performance constraints due to hardware limitations, and optimizing system configuration for various uses. Concepts and strategies are illustrated with case studies.

SCO Merge User's Guide
> describes how to use and configure an SCO® Merge™ system. Topics include installing Windows, installing DOS and Windows applications, using DOS with the SCO OpenServer operating system, configuring hardware and software resources, and using SCO Merge in an international environment.

SCO Wabi User's Guide
> describes how to use SCO® Wabi™ software to run Windows 3.1 applications on the SCO OpenServer operating system. Topics include installing the Wabi software, setting up drives, configuring ports, managing printing operations, and installing and running applications.

The SCO OpenServer Development System includes extensive documentation of application development issues and tools.

Many other useful publications about SCO systems by independent authors are available from technical bookstores.

Typographical conventions

This publication presents commands, filenames, keystrokes, and other special elements as shown here:

Example:	Used for:
lp or **lp**(C)	commands, device drivers, programs, and utilities (names, icons, or windows); the letter in parentheses indicates the reference manual section in which the command, driver, program, or utility is documented
/new/client.list	files, directories, and desktops (names, icons, or windows)
root	system, network, or user names
filename	placeholders (replace with appropriate name or value)
⟨Esc⟩	keyboard keys
`Exit program?`	system output (prompts, messages)
yes or **yes**	user input
"Description"	field names or column headings (on screen or in database)
Cancel	button names
Edit	menu names
Copy	menu items
File ➪ Find ➪ Text	sequences of menus and menu items
open or **open**(S)	library routines, system calls, kernel functions, C keywords; the letter in parentheses indicates the reference manual section in which the file is documented
$HOME	environment or shell variables
SIGHUP	named constants or signals
"adm3a"	data values
employees	database names
orders	database tables
buf	C program structures
`b_b.errno`	structure members

How can we improve this book?

What did you find particularly helpful in this book? Are there mistakes in this book? Could it be organized more usefully? Did we leave out information you need or include unnecessary material? If so, please tell us.

To help us implement your suggestions, include relevant details, such as book title, section name, page number, and system component. We would appreciate information on how to contact you in case we need additional explanation.

To contact us, use the card at the back of the *SCO OpenServer Handbook* or write to us at:

> Technical Publications
> Attn: CFT
> The Santa Cruz Operation, Inc.
> PO Box 1900
> Santa Cruz, California 95061-9969
> USA

or e-mail us at:

> *techpubs@sco.com* or *... uunet!sco!techpubs*

Thank you.

Office Automation

Chapter 1

Using SCO Shell

SCO Shell provides a menu-driven interface to the SCO OpenServer system. Using SCO Shell you can select the applications on your system from a single menu system, manage your files and directories, and run system utilities. SCO Shell is easier to use than the usual interface (or shell).

This chapter describes how you can use SCO Shell to manage your files and directories and run other utilities. It explains how to:

- start SCO Shell (this page)

- manage files with SCO Shell (page 16)

- use the Manager menu (page 19)

- run utilities and applications (page 43)

- print files (page 48)

SCO Shell also comes with two productivity tools: a calendar and a desktop calculator. These are described in Chapter 2, "SCO Shell accessories" (page 49).

Starting SCO Shell

To start SCO Shell, type **scosh** at the command prompt (on a character display terminal) or in a shell window (on the desktop). After a few moments the main SCO Shell screen appears. If you are unsure how to get to the command prompt, see "Getting to the command prompt" (page 80).

Your system administrator might have configured SCO Shell to start automatically when you log into your computer. In that case, you enter SCO Shell directly without stopping at the operating system first.

What the SCO Shell screen areas do

The following example shows the SCO Shell screen:

```
                          menu line                        context indicator
                                                                  |
               divider line              description line         |
                     |                        |                   |
                     |                        |                   |
  Application Manager Print Utility Options Quit              SCO Shell
  Select an application           |           |
  /u/perry                                  Thursday December 15, 1994 10:39
  ┌──────────────────────────────┐ ┌───────────────────────────────────┐
  │ Phone      - Send Messages   │ │           Dec 1994                │
  │ Mail       - Electronic Mail │ │    S  M Tu  W Th  F  S             │
  │ Calendar   - Event Scheduler │ │             1  2  3               │
  │ Word       - Microsoft Word  │ │    4  5  6  7  8  9 10             │
  │ System     - Run A System Shell│ │  11 12 13 14 15 16 17           │
  │                              │ │   18 19 20 21 22 23 24            │
  │                              │ │   25 26 27 28 29 30 31            │
  │                              │ │                                   │
  └──────────────────────────────┘ └───────────────────────────────────┘
  ┌──────────────────────────────┐ ┌───────────────────────────────────┐
  │ ../                          │ │ 11:00-12:00  Call Geneva          │
  │ mailfolders/    mymbox       │ │ 12:00                             │
  │ addresses       review       │ │  1:00- 2:00  Lunch appointment    │
  │ dead.letter                  │ │  2:00                             │
  │ ld.spec                      │ │* 2:30- 4:00  Sales meeting        │
  │                              │ │  3:00                             │
  │                              │ │  4:00                             │
  │                              │ │  5:00                             │
  └──────────────────────────────┘ └───────────────────────────────────┘

       File window                          Event window
                Application window                    Calendar window
```

The components of the screen are as follows:

- The *context indicator* appears on the status line and provides information about the SCO Shell screen that you are viewing.

- The *menu line* displays the menu items that are currently available. These are the actions that you can tell the SCO Shell to perform; they vary according to where you are in the menu system. In this example, the menu line displays the main menu for SCO Shell. The menu item **Application** is highlighted; you can pick a different menu item to be highlighted using the ⟨Left Arrow⟩ and ⟨Right Arrow⟩ keys, or the space bar.

- The *description line* gives a brief description of the highlighted menu item. If you press ⟨Enter⟩ to select the current highlighted item, this is what will happen. (When you move the highlight to a different menu item, the description line changes.)

- The *divider line* is the bar of text in reverse video that separates the menu and description lines from the display windows. The line shows your current working directory at the left, and the date and time at the right.

- The *display windows* for the SCO Shell screen include the *Application*, *Calendar*, and *File* windows, by default. You can change the windows that are on display; see "Changing the appearance of windows" (page 24).

- The *File window* contains a list of files (documents containing text, information, or programs) and directories stored in the current directory. (A directory is a storage area on your computer, like a drawer in a filing cabinet.)

- The *Application window* displays a list of applications (large programs) that SCO Shell knows about and can run for you.

- The *Calendar window* contains a one month rolling calendar.

- The *Event window* contains a list of meetings, appointments, telephone calls, and so on that you have scheduled for today. The asterisk at the left hand side indicates the next event. You control the events displayed in this window using the SCO Shell Calendar. See Chapter 2, "SCO Shell accessories" (page 49) for further details.

Using menus in SCO Shell

You interact with SCO Shell using the menu line displayed near the top of the screen. Move between the menu items using the ⟨Left Arrow⟩ and ⟨Right Arrow⟩ keys or the space bar. If you select a menu item and press ⟨Enter⟩, the associated action is carried out. Alternatively, you can select a menu item by pressing the key corresponding to the first letter of its name.

Some menu items change the menu line to ask you for more information. For example, if you select **Options**, the menu line changes to display a different set of options. You can go back to the previous menu by pressing ⟨Esc⟩.

Other menu items present you with a list of options otherwise known as a *point-and-pick* list. For example, if you select **Application**, the cursor moves to a new window below the menu bar; you can move up and down the list of available applications using the ⟨Up Arrow⟩ and ⟨Down Arrow⟩ keys. You can run an application by pressing ⟨Enter⟩.

Canceling an operation

Pressing ⟨Esc⟩ usually allows you to leave a menu that you have entered by mistake, or to stop a process or an operation for any other reason. When you press ⟨Esc⟩, you return to wherever you were before you started the operation.

SCO Shell does not allow you to use ⟨Esc⟩ to leave Mail, Calendar, the Application or Utility List editors, and certain other menus. You must select **Quit** from the menu line or press ⟨F2⟩ to exit these. You are asked to confirm that you wish to leave, and you may also be given the option of saving or abandoning any changes you have made.

Error messages

If you ask SCO Shell to do something that cannot be done, for example, to copy a file that does not exist (by mistyping the filename, for instance), an error message appears at the bottom of the screen. This message should give you the information you need to correct the problem. After reading the error message, press ⟨Enter⟩ to continue.

Getting help in SCO Shell

The SCO Shell can provide you with screens of help information if you get lost. You can get help in any situation by pressing ⟨F1⟩. This provides you with a summary of how to use help. Press ⟨F1⟩ again to enter the help system. A window appears containing help text. You can select options from the menu at the top of the screen for additional help, or an index of help topics. To leave help and return to whatever you were doing, press ⟨Esc⟩ or select **Quit** from the menu.

Using the accelerator keys

There are a set of accelerator keys available to you. Depending on your terminal type, you may see these keys listed at the bottom of your screen.

Key name	Key	Action
Help	⟨F1⟩	Calls up help screens that explain the feature that you are using.
Quit	⟨F2⟩	Quits quickly from any part of SCO Shell. From the main menu, it quits SCO Shell.
List	⟨F3⟩	Displays special point-and-pick lists of reference information.

(Continued on next page)

(Continued)

Key name	Key	Action
Spell	⟨F4⟩	Spell checks a single word.
Search	⟨F5⟩	Searches for an entry in any point-and-pick list.
Calendar	⟨F6⟩	Displays the Calendar window; if the Calendar window is already displayed, it displays past or future months in the Calendar window. Use the ⟨Right Arrow⟩ or ⟨Left Arrow⟩ to move forward or backward a month at a time.
Enter	⟨F10⟩	Sends a command or data to the operating system or to an application.

The task you are performing determines which of these keys are available.

Using a mouse

You can use a mouse with SCO Shell if you are working at the console and the appropriate hardware has been connected to your system.

SCO Shell expects you to have a three-button mouse (although you can work with a two-button mouse as well). Mouse buttons are numbered from left to right. You can move around the screen area freely; when the mouse cursor is positioned over an item, you can press one of the buttons to achieve the following effects:

Button 1 Equivalent to pressing ⟨Space⟩; typically moves to the next item in the current group. For example, if you press button 1 while the mouse cursor is on the menu, the cursor jumps to the next item; if you press button 1 while the mouse cursor is on a window containing a list, the window scrolls up or down (depending on whether the cursor is in the upper half or lower half of the window).

Button 2 Equivalent to pressing ⟨Enter⟩; typically activates the current item. For example, if you press button 2 while the mouse cursor is on an application in the **Application** list window, that application is activated.

Button 3 Equivalent to pressing ⟨Esc⟩; typically aborts the current operation or returns to the previous level of a menu.

If you have a two-button mouse, you lose Button 1's functionality; use the ⟨Space⟩ key instead.

Quitting SCO Shell

To quit SCO Shell, go to the top level menu. Select the **Quit** option, and press ⟨Enter⟩. SCO Shell asks you whether you want to quit and offers you the option of saying yes or no (if you say no, SCO Shell returns you to the main menu). Alternatively, press ⟨F2⟩ in any menu.

Managing files with SCO Shell

SCO Shell allows you to manage and organize your files without resorting to complex operating system commands. Using the options on SCO Shell's **Manager** menu, you can:

- copy, rename, and remove files

- create and remove directories

- change the permissions on your files and directories

- save and retrieve files from disk or tape

- access the clipboard

Files and directories

Computers store information in files. When you use an application, it generally creates a file to hold your work; this is one way files are created.

Every file has a name and some contents. A file usually contains some piece of information, such as a letter, report, or phone list. The filename is a label you give the information to keep track of it.

A computer can contain thousands of different files. To manage this huge group of files, the operating system groups them into directories. Files that belong to a certain person or files associated with a particular program are often stored in a directory of their own. To see a particular file, you have to go to the directory that holds it. Each directory has a name, just like a file. However, while a file holds information, a directory holds files and other directories, known as subdirectories.

When you log into your computer, you are in your home directory. This is where you keep your own files and do your work. The name of your home directory is probably the same as your login.

Using subdirectories

After a while, files start to accumulate in your home directory. To keep things organized, you can create some subdirectories within your home directory. If you keep several different types of files in your home directory, you can create subdirectories for each type and divide your files among them. Then, when you want to work with a particular group of files, simply go to the subdirectory that holds them.

For example, you might create a *letters* subdirectory in which to keep your letters, and separate subdirectories for particular projects you are working on. You can further divide your new subdirectories. For example, you can make new subdirectories in your *letters* directory to organize your letters by person or subject. There is no limit to the number or kinds of subdirectories you can create.

Every directory (except one) is a subdirectory of some other directory. The directories are organized into an inverted tree structure, so called because directories branch out of other directories like the branches of a tree. This tree is "inverted" because the branches move down, not up. One directory, at the top of the tree, is not a subdirectory of any other directory. This is called the *root* directory, and its name is a slash character (/). To help clarify this, look at the following picture of part of a typical directory tree. (Note that directories have small sub-trees below them; files do not.)

/ (root)

etc   bin   u   dev

jeffa   calvinh   perry   graceh

dead.letter   review   mailfolders   mymbox
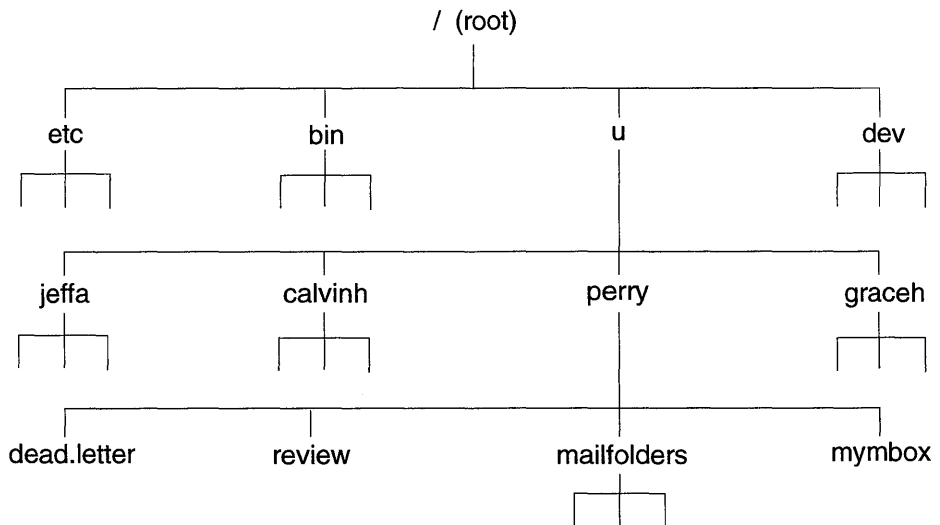
Notice that directories have lines leading to files or other directories. Files do not have any lines coming out of them because they do not lead to other files or directories. Remember that this is only a small portion of the entire directory tree. In reality, most directory trees are much deeper than four levels, and each level contains many more than four files and directories. Some directories may also not contain any files or directories.

Pathnames

Any file or directory on the computer can be identified uniquely by its pathname. A pathname is like a map with directions for finding a file or directory; it lists, in order, each directory you must pass through to get from the *root* directory to the file or directory in question. When the pathname is written down, the directories are separated by slashes (/). Remember that the slash character is also the name of the *root* directory; the first slash in a pathname stands for the *root* directory, while the others are used to separate directories.

For example, the full path of the file called *review* in the diagram above is */u/perry/review*. The pathname tells you that *review* is in the directory called *perry*, *perry* is a subdirectory of *u*, and *u* is in the *root* directory.

A pathname that begins at the *root* directory is called an absolute pathname or full pathname. Pathnames that begin at some level below the *root* directory, called relative pathnames, are also useful. When you work in one directory, you can specify a file or directory below it by its relative pathname. For example, if you are working in the directory called *u* and you need to specify the file in *perry* called *mymbox*, you can use the relative pathname *perry/mymbox*.

The current directory

The directory you are working in at the moment is called your current directory. When you first log into your computer, your home directory is your current directory. Whenever the **Manager** menu is displayed on your screen, the files and subdirectories in your current directory are also listed. You can change your current directory at any time. (This process is described in "Changing the current directory" (page 35).)

Naming and organizing files and directories

Here are a few important rules to follow when you name files and directories:

- No two files in the same directory can have the same name.

- Do not use blank spaces. (To represent a space in a name, use an underscore character (_), or period (.), instead.)

- File names may be limited to a maximum of 14 characters on some systems. (If you are not sure, ask your system administrator.)

- Do not use control characters in filenames. (Control characters are keys pressed while holding down the ⟨Ctrl⟩ key.)

- Do not use any of the following characters in filenames:

  ! " ´ ´ ; / \ $ * & < > ( ) | { } [ ] ˜

  These characters are reserved for operating system commands.

It is a good idea to use filenames that describe the contents of the files and are easy to type. For example, a file containing a letter to your friend Bob would be more appropriately called *bob.letter* than *xxx* or *5il%ds*, although all of these are acceptable filenames to the system.

It is very easy to let your files pile up in one directory until it is difficult to find anything. Always create more subdirectories when you need them.

The **Manager** menu options make it easy to create and remove directories, remove files, and to move up and down the directory tree. These operations and others are described later in this chapter.

Using the Manager menu

To get to the **Manager** menu, select the **Manager** option from the SCO Shell menu. The **Manager** menu appears at the top of the screen:

```
                                                              Manager
View Edit File Directory Archive Transfer Preferences Quit
Displays a file
/u/perry                                Wednesday February 22, 1995 11:27
```

If you are unfamiliar with the parts of the screen, see "What the SCO Shell screen areas do" (page 12). For an explanation of how to alter the screen display, see "Changing the appearance of windows" (page 24).

The tasks that you can perform are described in detail in "Using the Manager menu options" (page 23). Many of the tasks that the Manager can perform involve two basic steps: first you specify the action to be done, (by selecting a menu option), and then you select the file or files you want to be acted upon.

Selecting files

Many of the operations that you can perform with the **Manager** menu options require you to select the files to operate on. For example, if you select **Copy**, you must tell SCO Shell what file to copy.

Choosing files

There are three ways to pick a file:

- type the filename

- use the arrow keys to move the highlight to the filename in the point-and-pick list

- if you are looking through a list, you can use the Search key ⟨F5⟩

When you have found and selected the file by one of these methods, press ⟨Enter⟩ to execute the selection.

You can also select several files at a time. If you are selecting files via the point-and-pick list, move the highlight to each file you want and press the ⟨Space⟩ bar. Each time you do this, an asterisk (*) appears in front of the filename. When you have finished, press ⟨Enter⟩ to select every file marked with an asterisk. In this example, the files *notes.old* and *notes.new* have been selected:

```
██████████████   * notes.old
  letter.new       poster.new
  letter.old       poster.old
* notes.new
```

For more information on point-and-pick lists, see "Using menus in SCO Shell" (page 13).

If you are typing in names rather than pointing and picking, selecting several files at a time is performed differently. To select three different files for an operation, you cannot enter three separate filenames at once; SCO Shell accepts only one at a time. Instead, you can type instructions telling SCO Shell to display all files whose names share certain characters. This technique is described in "Using wildcard characters" (page 21).

If you want to find a file or a number of files, select:

Manager ⇨ **File** ⇨ **Find**

and specify a filename or part of a filename to search for. The file(s) that match the name that you specify appear in a list, and you can choose one or more files from this list for the option you are working with.

Changing directories

If the file you want is in a subdirectory of the current directory, you must first select that directory in the same way as you select files. Directories are distinguished from files in the listing on the screen because a directory is always followed by a slash (/).

After highlighting the directory, select it either by pressing ⟨Enter⟩ or by typing the "＞" character. A list of the directory's files then appears on the screen. Now select the file(s) you want. When you finish working, your home directory listings reappear.

If you need to move up to a directory above yours (as from */u/perry* to */u*), type the "＜" character. You can also select the symbol for the next directory up, "../" (dot dot slash), that appears at the beginning of every list of directories and files.

To move to a directory that is not in the current directory or in the directory above it, select:

Manager ⇨ **Directory** ⇨ **Change**

to search for and move to a directory with a specified name. For more information on this command, see "Managing directories" (page 34).

Using wildcard characters

SCO Shell understands two wildcard characters that make selecting files more efficient. A wildcard is a character that, when used in a filename, matches some other character or string of characters.

For example, suppose the current directory contains the files listed below:

```
* ../
  ch1.start.dcx
  ch2.run.dcx
  ch3.repairs.dcx
  ch3.repairs.psf
```

You can refer to the file *ch2.run.dcx* by selecting it, or by typing in its whole name. But you can also refer to it using the asterisk wildcard, as *ch2.*. An asterisk is a wildcard that matches any sequence of characters; it could equally well match *.dcx* or *.psf* or *run.dcx*, but because there is only one file in the directory that begins with *ch2.*, that is the file which is matched.

If you try selecting *ch3.r*, SCO Shell will select two files for you: *ch3.repairs.dcx* and *ch3.repairs.psf*. This is because the two filenames both start with *ch3.r*, and the asterisk matches all possible suffixes.

* on its own matches every file in the directory. All the files begin with *ch*, therefore *ch* also matches all the files.

Matching an unlimited run of characters is not always necessary. Suppose we want to match the files ending in *.dcx* and with a middle part beginning with *r*. We could use the pattern *.r*.dcx*. However, this is imprecise. (In a directory containing many files, it might return spurious matches.) The question mark (?) is a wildcard that substitutes for any single character. For example, *ch?.r*.dcx* matches the characters *ch*, followed by a single character, then a period, then a run of characters terminated by *.dcx*. This matches *ch2.run.dcx* and *ch3.repairs.dcx*, but could not match a file called, for example, *ch10.routing.dcx* (which contains two characters between *ch* and *r*.

When you have selected a subset of files using a wildcard character, you can select from the subset using the ⟨Space⟩ bar, and then pressing ⟨Enter⟩ to carry out the functions you require.

Using the Manager menu options

The following sections discuss the tasks that you can perform with the options on the **Manager** menu.

Looking at a file

You can display the contents of a file on your screen with the **View** option. When you select **View**, a listing of the files and subdirectories in your current directory appears. Here is an example:

```
                                                              View

 Enter the file(s) to view: /u/perry/
/u/perry                                    Thursday December 15, 1994 10:39
 ┌────────────────────────────────┐  ┌────────────────────────────────────┐
 │  Phone        - Send Messages  │  │            Dec 1994                 │
 │  Mail      .  - Electronic Mail│  │    S  M Tu  W Th  F  S              │
 │  Calendar     - Event Scheduler│  │             1  2  3                 │
 │  Professional - Spreadsheet    │  │    4  5  6  7  8  9 10              │
 │  Word         - Microsoft Word │  │   11 12 13 14 15 16 17              │
 │  News         - Read or Send News │ 18 19 20 21 22 23 24              │
 │  System       - Run A System Shell │ 25 26 27 28 29 30 31             │
 └────────────────────────────────┘  └────────────────────────────────────┘

 ┌─────────────────────────┬───────────────────┬──────────────────────────┐
 │ * ../                   │ News/             │ UNIX3-2-5/               :│
 │   APPRAISAL/            │ OBJECTIVES/       │ USERSURVEY/              │
 │   ARCHITECTS/           │ ONLINE/           │ VDISK/                   │
 │   BOOKBUILD/            │ OP_FILES/         │ VDM_GRAPHICS/            │
 │   COVERLETTERS/         │ PERF_GRAPHICS/    │ VIDEO/                   │
 │   DESIGN/               │ PUBLISHING/       │ bin/                     │
 │   DOCHOME_DESIGN/       │ PUBLISHINGSYSTEM/ │ clipdir/                 │
 │   DUMP/                 │ QRC/              │ graphics/               v│
 └─────────────────────────┴───────────────────┴──────────────────────────┘
```

Now you can select the file(s) that you want to view. If you are not sure how to do this, see "Selecting files" (page 19).

In the example above, the File window covers roughly half of the screen. Your main SCO Shell screen, however, might be configured for a larger or smaller File window, that is, one that fills all of the screen or a quarter of it. See "Changing the appearance of windows" (page 24) for a description of how to customize the screen layout.

If there are more files than can appear in the window at one time, use the ⟨Down Arrow⟩ key to scroll the additional listings into the File window. You can also use the ⟨PgUp⟩ and ⟨PgDn⟩ keys to move up and down through the listings one "page" at a time. In addition, the ⟨Home⟩ and ⟨End⟩ keys move you to the beginning and end of the file listings, respectively.

After you make your selection, the first screen of text appears. In the following example, the file *mymbox* is being viewed:

```
                  Currently viewing /u/perry/mymbox
                Arrow keys to scroll window, <esc> when finished
/u/perry                                Wednesday February 22, 1995    9:25
From charles Tue Feb 21 9:06:13 1995
To: perry
Subject: interface design spec review
Date: Tue Feb 21 9:09:47 1995

The review is going well, Perry - I'll
definitely be done by the end of the month.

-Charlie

        From perry Fri Feb 17 14:31:16 1995
        To charles
        Subject: interface design spec review
        Cc: rogerm perry
        Date: Fri Feb 17 14:32 1995

        Have you had a chance to look at that review yet?
        Do you think the due date of March 1 is reasonable?

        -Perry
```

You can now scroll through the file to read it. These are the keys that you can use with **View**:

Key	Action
⟨Esc⟩	exits the file
⟨Up Arrow⟩	moves up one line
⟨Down Arrow⟩ or ⟨Space⟩	moves down one line
⟨PgDn⟩ or ⟨Enter⟩	moves down a page
⟨PgUp⟩	moves up a page
⟨Home⟩	moves to the top of the document
⟨End⟩	moves to the bottom of the document

Press ⟨Esc⟩ when you finish viewing the file. If you select more than one file to view, the first screenful of the next file is displayed. Otherwise, you return to the SCO Shell menu.

Changing the appearance of windows

You can change the relative position and size of the information windows on the SCO Shell display. Select **Options** ▷ **Display** then either **FileWindow, AppWindow, CalendarWindow,** or **EventWindow** to identify the window you want to modify.

All these selections present you with a simple menu; there is some additional information to fill in for the Applications or File windows. The basic information you specify is where you want the window to appear on the display (there are a number of predefined options), and how large to make it. A window may occupy a numberered quarter (1 for top left, 2 for top right, 3 for bottom left, and 4 for bottom right), the top half, or the bottom half of the display.

In addition, the **Applications** menu has an option to display either brief or detailed information about the available programs.

Changing the content of the File window

In addition to the basic information, the File window has options to select the style of listing, the sort order, and whether to sort the files by date or by name. If you choose a brief style listing, only filenames are displayed; the system style listing gives you a full view of filenames with their corresponding owner and group IDs, permissions, size, and creation time. The following is an example of a system style listing:

```
* bin/            drwxr-xr-x  2 perry   tech      48  06/18/95  4:50   ^
  clipdir/        drwxr-xr-x  2 perry   tech      32  08/12/95  2:44   |
  mailfolders/    drwxrwxrwx  4 perry   tech     144  06/15/95  2:32   |
  training/       drwxr-xr-x  2 perry   tech    2240  08/10/95  2:10   :
  wastebasket/    drwxr-xr-x  2 perry   tech      32  08/11/95  3:04   |
  wp6/            drwxrwxrwx  3 perry   tech      48  06/15/95  2:33   |
  Booklist        -rw-r--r--  1 perry   tech    8299  06/28/95  1:27   |
  DCEsched        -rw-r--r--  1 perry   tech   22697  07/01/95  9:24   v
```

Note that the Event window has been turned off and the File window expanded to the full width of the screen.

Editing a file

You can edit the contents of a text file using the **Edit** option in the **Manager** menu. When you select **Edit**, a listing of the files in your current directory appears, as explained in the section, "Looking at a file" (page 23).

Select the file(s) that you want to edit. Once you select a file, the first screen of text appears.

Editing commands

To edit a text file, use the Edit Mode commands available for editing text in form fields. The following table gives a brief introduction to the editing commands.

Command	Description
Arrow keys	Moves the cursor up and down through the file, and to the right or left on each line.
⟨PgUp⟩ or ⟨PgDn⟩	Use ⟨PgUp⟩ to jump back to the previous page, or ⟨PgDn⟩ to jump forward to the next page. A "page" is one screen of lines.
⟨Home⟩	Moves the cursor to the beginning of the current line.
⟨End⟩	Moves the cursor to the end of the current line.
⟨Esc⟩	Switches between editing the file and using the **Edit** menu.
⟨Ctrl⟩G↑	Moves to the top of the document.
⟨Ctrl⟩G↓	Moves to the bottom of the document.
⟨Ctrl⟩N	Moves the cursor to the next word.
⟨Ctrl⟩P	Moves the cursor to the previous word.
⟨Ctrl⟩W	Deletes the word the cursor is on, if the cursor is on the first character of the word.
⟨Ctrl⟩Y	Deletes the current line (the line the cursor is on).
⟨F8⟩	Pastes a deleted line back into the file. The Edit program stores the last 10 lines you deleted. When you press ⟨F8⟩, the last line you deleted reappears. If you press ⟨F8⟩ again, the next to last line you deleted reappears, and so on.
⟨Ctrl⟩O	Inserts a blank line above the current line.
⟨Ctrl⟩V	Switches between Insert and Overstrike mode. You are in Insert mode to start. If you move the cursor to some existing text and begin typing, the new words are inserted between the existing words. If you press ⟨Ctrl⟩V to change to Overstrike mode, the words you type replace (overstrike) any existing words on the same line. Press ⟨Ctrl⟩V again to return to Insert mode.
⟨Ctrl⟩Z	Calls up the pop-up list of extra file-editing and formatting options. This is the same as pressing ⟨F9⟩.
⟨F5⟩	Prompts you to enter a word, then searches for the next occurrence of that word in your file.

Pop-up list of extra edit-mode commands

When you press ⟨F9⟩ in Edit mode, a list of extra editing commands appears on the screen, like this:

```
┌──────Extra Commands──────┐
│* Format Paragraph        ┃
│  Join Lines              │
│  Margin (set)            │
│  Delete Line (^y)        │
│  Paste (undelete, <f8>)  │
└──────────────────────────┘
```
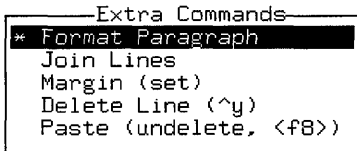
To choose a command from this list, position the highlight bar over the command that you want and press ⟨Enter⟩. To use the **Format Paragraph**, **Delete Line**, and **Join Lines** commands, you must position the cursor on the line or paragraph that you want before pressing ⟨F9⟩.

The following table describes the commands on the Extra Commands list:

Command	Description
Format Paragraph	Reformats the paragraph that the cursor is on. This command fills each line with text out to the margin, moving text up from subsequent lines as necessary.
Join Lines	Joins the line the cursor is on with the text of the next line. If both lines together are too long to fit within the file margin, only part of the text from the second line is joined to the first.
Margin (set)	Resets the line length for this file. The default line length appears in a box. Type in the new line length and press ⟨Enter⟩. If you type in new text now, the text automatically breaks to a new line after reaching the line length that you set. To reformat existing paragraphs to the new line length, use the **Format Paragraph** command. To reset the default line length for all editing, use the **Preferences** command in the **Manager** Menu.
Delete Line (^y)	Deletes the line that the cursor is on. This option is the same as pressing ⟨Ctrl⟩**Y**.
Paste (undelete, ⟨F8⟩)	Pastes deleted lines back into the file. When you choose **Paste**, the last line that you deleted reappears. If you choose **Paste** again, the next to last line that you deleted reappears, and so on, up to a maximum of 10 lines. Choosing **Paste** is the same as pressing ⟨F8⟩.

From within the editor, other functions are available; to use these other functions, press ⟨Esc⟩ while editing a file. The **Edit** menu appears, with the following options:

```
                            <esc> to resume editing                    Edit
Edit  Include Transfer Quit
Edit document
/u/perry                                    Wednesday February 22, 1995   3:01
```

The options in the menu are described briefly in the following table:

Option	Description
Edit	invokes the Auto Editor
Include	includes a file in the current document
Transfer	moves items to and from the Clipboard, and deletes items from the Clipboard
Quit	optionally saves your work, exits the editor, and returns to the **Manager** menu

If you choose the **Edit** option, the text of the file appears, and you can edit it using the Auto Editor. When you access the editor, you will, by default, use a built-in editor; however should you wish to use a different editor, you can configure your system so that you can use your own favorite editor. This is known as the Auto Editor; you can choose an Auto Editor by selecting: **Manager ⇨ Preferences** .

Select the **Include** option to include another file in the file that you are currently editing. When you select **Include**, a pop-up window appears listing the files in the current directory. Use the cursor movement commands to select a file. The file is inserted after the cursor position.

The **Transfer** option allows you to access the Clipboard while you are editing a file. You can use the Clipboard to transfer text and other forms of data between different applications. See "Using the clipboard from the Manager menu" (page 41) for more information about using the Clipboard.

The **Quit** option exits the Edit Mode and returns you to the **Manager** menu. **Quit** also saves your latest changes to the current file.

Managing files

Use the **File** option in the **Manager** menu to organize and manipulate files. When you select the **File** option, the following menu appears:

```
                                                                       File
Copy Rename Erase Find Unfind Permissions WasteBasket
Copies a file
/u/perry                                    Wednesday February 22, 1995   3:03
```

Here are brief descriptions of the options in the **File** menu:

Option	Description
Copy	makes a copy of a file or a group of files
Rename	changes a file's name, or moves a file or group of files to a different directory
Erase	puts one file or a group of files in the wastebasket
Find	lists files that match specified criteria
Unfind	lists files in the current directory, rather than the "Find" list
Permissions	changes the permissions on your files
WasteBasket	recovers or removes files from the wastebasket

The following sections discuss the tasks that you can perform with the options of the **File** menu.

Copying files

Use the **Copy** option to make a copy of one or several files at a time. You can place the copy in any directory for which you have the correct permissions. For more information on file permissions, see "Changing permissions" (page 32).

After you select the **Copy** option from the **File** menu, a list of the files in your current directory appears, along with a message asking you to select the file that you want to copy.

Choose the appropriate file. When you make a selection, another message appears. This one asks you to select the file to copy to. Now you must select a destination for your file(s).

If you are copying a single file, you can enter any destination filename that does not already exist. Do not enter the name of a file that already exists; doing this overwrites the contents of the file, replacing them with the new data. If you specify a directory only, a copy of your file with its original name is put in that directory. If you are copying a group of files, you must specify a directory as a destination.

Examples

This first example creates a backup copy of the file *review* called *review.bak*. First, select the **Copy** option. The **File** menu disappears, and you are asked to enter the name of the file from which you wish to copy. In this case, type **review**. The word appears after the pathname */u/perry*.

Now press ⟨Enter⟩ to execute the selection. At this time, you are asked to enter the name of the file to which you want to copy. In this case, type **review.bak** and press ⟨Enter⟩. The **File** menu appears. There is now a file called *review.bak* in the directory */u/perry*, containing an exact copy of the file *review*.

Here is another example. This time, copy the files *dead.letter* and *mymbox* into the directory *mailfolders*. Again, you must select the **Copy** option. Now mark the files to be copied by moving the highlight to each file and pressing ⟨Space⟩. See "Selecting files" (page 19) if you are not sure how to do this. This is what the screen looks like after you mark one of the two files:

```
                                                                    Copy
   Enter the file to copy from: /u/perry/mailfolders/
 /u/perry/mailfolders                        Thursday August 3, 1995  3:20
  ┌──────────────────────────────┐  ┌──────────────────────────────────┐
  │  Phone        - Send Messages │  │         Aug 1995                 │
  │  Mail         - Electronic Mail│ │ Tu  W Th  F  S  S  M             │
  │  Calendar     - Event Scheduler│ │  1  2 ▓  4  5  6  7              │
  │  Professional - Spreadsheet    │ │  8  9 10 11 12 13 14             │
  │  Word         - Microsoft Word │ │ 15 16 17 18 19 20 21             │
  │  News         - Read or Send News│ 22 23 24 25 26 27 28             │
  │  System       - Run A System Shell│29 30 31                        │
  └──────────────────────────────┘  └──────────────────────────────────┘
  ┌───────────────────────────────────────────────────────────────────┐
  │  ../           dead.letter     lenore.notes    review              │
  │  mailfolders/  id.spec      * mymbox           review.bak          │
  │                                                                     │
  │                                                                     │
  │                                                                     │
  └───────────────────────────────────────────────────────────────────┘
```

Now, press ⟨Enter⟩ to execute the selection. The "copy to" line appears, just as in the previous example. Type **mailfolders** or select *mailfolders* on the screen to select the *mailfolders* directory. When you press ⟨Enter⟩ copies of the two files are placed in the *mailfolders* directory, and you return to the **File** menu. SCO Shell gives the copies the same names and permissions as the original files.

Renaming and moving files

The **Rename** option is used both for changing the name of a file and for moving files from one directory to another. It can also be used to rename directories. Using the **Rename** option is similar to using the **Copy** option, but you do not leave a copy of the file in the original location. After selecting the **Rename** option you perform the same steps to rename a file as to copy a file, and the procedures involved in moving files to a different directory are just like those used to copy files to a different directory.

When you select the **Rename** option from the **File** menu, a message appears, prompting you to select the file that you want to rename.

When you have made your selection, a new message appears, prompting you to enter the new filename. At this point, you select a destination for your file(s).

- If you are renaming a single file, you can specify any filename; the file is then renamed (that is, moved to a new filename). Note that if a file of that name already exists, it will be replaced by the new file.

- If you specify a directory, your file moves into that directory, keeping its original name. If you are renaming a group of files, you must specify a directory as a destination.

Removing files

To remove files, use the **Erase** option; this option does not remove directories. (See "Managing directories" (page 34) for information on removing a directory.) When you select the **Erase** option, you are prompted to select the file that you want to erase. Once you select the files and press ⟨Enter⟩, you return to the **File** menu. The directory listing is updated to show that the files have been erased.

When you erase a file, SCO Shell places it in your wastebasket directory. The file remains there until you exit SCO Shell. You can recover files from the wastebasket with the **Wastebasket** option, see "Recovering erased files" (page 34). When you exit SCO Shell, all the files in the wastebasket directory are permanently deleted, and there is no way to recover them.

Finding files

The **Find** option lets you search for and list files with specified names. **Find** looks for a named file starting in the current directory, then in all sub-directories. When you select the **Find** option, a form appears, with a field for specifying the full or partial filename that you want to search for. The form looks like this:

```
┌─────────────── File Search ───────────────┐
│                                            │
│  Filename or partial name: [            ]  │
│                                            │
└────────────────────────────────────────────┘
```

Use the full filename if you want to find just one file. If you want to find a number of files with similar names, specify the part of each filename that is the same, and use wildcard characters for the rest of the filename. For example, to list all files with the *.dcx* extension, enter *.dcx** in the "Filename" field. For more information about using wildcard characters, see "Using wildcard characters" (page 21).

Note that the listing produced might contain files from numerous different directories at different levels of your directory tree. If you no longer want to see the listing of files from different directories, and instead you want to display only the files in your current directory, choose the **Unfind** option in the **File** menu. After choosing **Unfind**, you can no longer see files in numerous different directories in your directory tree.

You can use the **Find** option to carry out a task on several different files in a list. First use the **Find** option to display the list of files. Choose a task option from the **File** menu. Finally, choose the files to be acted upon.

Managing file permissions

Every file or directory has a set of permissions that control who can read them, change them, and execute them. With the **Permissions** option, you can change permissions on files and directories you own to make them more or less accessible to other users on your system.

How permissions work

Each file has three different sets of permissions: one for the user, the owner of the file; one for the group, the other users in your work group; and one for all, meaning everyone on your system. Each set of permissions can include none, one, or more than one of the following privileges:

Read   For a file, this means you can look at its contents on the screen. For a directory, this means that you can see a list of the files it holds. Read permission for the directory does not automatically give you read permission for every file in the directory.

Write   For a file, this means that you can alter its contents or remove it. For a directory, this means you can create files and subdirectories within that directory.

Execute For a file, this means you can run a program that the file contains. For a directory, this means you can change to and use files within that directory.

When you create a file or directory, it acquires a standard set of permissions automatically. For example, a typical configuration might give the *user* (yourself) read and write permissions. Other users, both your *group* members and *all* other users, might have only read permission.

Changing permissions

The **Permissions** option lets you change the permissions on one file at a time. (To change permissions on a directory use the

   Manager ⇨ **Directory** ⇨ **Permissions**

menu.) The **Permissions** option also shows you the current permissions.

To change permissions on a file, proceed as follows:

1. Select the **Permissions** option from the **File** menu. In response, this message appears:

```
                                                           Permissions
   Enter the file to be changed: /u/perry/mailfolders/
   /u/perry/mailfolders                     Wednesday February 22, 1995   3:23
```

2. Now, select the file you want to change. Remember, you can select only one at a time on this form. The File Permissions form now pops up on the screen:

```
   ┌──────────── File Permissions ──────────┐
   │                                        │
   │  User    Read[*]  Write[*]  Execute[ ] │
   │  Group   Read[*]  Write[ ]  Execute[ ] │
   │  All     Read[*]  Write[ ]  Execute[ ] │
   │                                        │
   │  Owner: [perry    ]                    │
   │  Group: [tech     ]                    │
   └────────────────────────────────────────┘
```

The form displays the current permissions for the file you selected, and the owner and owner's group. The asterisks between the brackets show who has what permissions. In this example, the User (owner) has Read and Write permissions, while Group and All have only Read permissions. This means that anyone can read the file, but only the owner can change it. No one can execute it.

3. To grant additional permissions for User, Group, or All, move the highlight to the appropriate permission field and press the ⟨Space⟩ bar. An asterisk shows that you have added this permission. Use the ⟨Up Arrow⟩, ⟨Down Arrow⟩, and ⟨Enter⟩ keys to move around on this form.

To remove permissions press the ⟨Space⟩ bar in a field that already has an asterisk: the asterisk disappears.

4. You can change the values in the "Owner" and "Group" fields if you are the owner of the file.

5. When you have finished, press ⟨Enter⟩ in the last field ("Group") or ⟨F10⟩, or simply press ⟨Ctrl⟩**X** with the cursor at any point on the screen. This makes the changes to the file or directory that you requested, and then returns you to the **File** menu.

Viewing permissions without changing them

If you just want to see what a file's permissions are, select it through the **Permissions** option but make no changes. Simply press ⟨Esc⟩ when you have finished looking at the File Permissions form.

Your File window can also show you a file's permissions if the window is configured to show system file listings; that is, listings that show detailed technical information about each file. See "Changing the content of the File window" (page 25) for details.

Recovering erased files

The wastebasket option lets you manipulate the files in your wastebasket directory. When you select the **Wastebasket** option, this menu appears:

```
                                                              WasteBasket
Select Delete Clear
Recovers a file from wastebasket
/u/perry/mailfolders                        Wednesday February 22, 1995  3:27
```

Here are brief descriptions of the **Wastebasket** menu options:

Option	Description
Select	recovers file(s) from the wastebasket
Delete	permanently erases file(s) from the wastebasket
Clear	permanently erases all of the files in the wastebasket

If you choose the **Select** or **Delete** option, a list of files in the wastebasket appears on the screen. You can then select the file(s) that you want to recover or delete. If you are not sure how to do this, see "Selecting files" (page 19).

> **NOTE** Any file that you recover from the wastebasket appears in your current directory. If you have changed directories since removing the file, it does not appear in the directory where it was before.

If you choose the **Clear** option, all the files in the wastebasket are permanently removed. The wastebasket is cleared automatically each time you exit SCO Shell.

Managing directories

The **Directory** option in the **Manager** menu lets you take a number of actions on directories. When you select the **Directory** option, the following menu appears:

```
                                                                 Directory
Change Make Remove Permissions
Changes current directory to another directory
/u/perry/mailfolders                        Wednesday February 22, 1995  3:27
```

The options on this menu allow you to change to a new current directory, make new directories, remove empty directories, and change directory permissions. These operations are discussed in the following sections.

Changing the current directory

To change your current directory, select the **Change** option from the **Directory** menu. In response, a message appears, prompting you to select the new current directory. Also, the File window changes to show only the subdirectories in your current directory; normal files are not displayed.

Now select the directory you want to work in. You can select a subdirectory from your current directory, or you can specify a different directory by its pathname. You return to the **Manager** menu, and the listing for the new current directory appears on the screen.

Creating a new directory

To create a new directory, select the **Directory** option from the **Manager** menu, and then select the **Make** option from the **Directory** menu. You are now asked to enter the name of the new directory.

Type the name of the new directory, and press ⟨Enter⟩. You return to the **Manager** menu, and the new directory appears in the directory listing displayed on the screen.

Removing an empty directory

Before you can remove a directory, you must make sure that it does not contain any files or subdirectories. If it is not empty, you must either delete the files it holds or move them to other directories.

Once the directory is empty, remove it by selecting the **Directory** option from the **Manager** menu. Then select the **Remove** option from the **Directory** menu; you are prompted to select the directory to remove.

When you have selected the directory and pressed ⟨Enter⟩, the directory is removed from the directory listing displayed on the screen, and you return to the **Manager** menu.

Changing directory permissions

If you choose the **Permissions** option in the **Directory** menu, you can change the permissions for the current directory (and subdirectories within the current directory) using the Directory Permissions form. For more information on this form and how to use it, see "Changing permissions" (page 32).

Copying files to and from tape or disk

The **Archive** option on the **Manager** menu provides an easy way to copy files to and from backup media such as disks and tape. These media are removable so that you can retain a personal archive copy of your work to transfer to another machine or to keep in a secure place.

Archive allows you to:

- extract files from a disk or tape and place copies into your current directory (extracting everything in the archive, or just specific files)

- display the files currently on the disk or tape

- format a disk before archiving files on it

- create an archive, that is, copy files from your current directory onto a disk or tape

When you choose the **Archive** option, the following menu appears:

```
                                                          Archive
Extract List Format Create Type Device
Retrieve files from a disk or tape
/u/perry/mailfolders                   Wednesday February 22, 1995  3:29
```

Here are brief descriptions of the **Archive** menu options:

Option	Description
Extract	copies files from disk or tape into your current directory
List	lists all files on a disk or tape
Format	formats a disk or tape to prepare it for copying files
Create	copies files from your current directory to a disk or tape
Type	selects the archive file format: **tar**, **cpio**, or DOS
Device	specifies the address of the disk or tape drive for archiving

Preparing to use removable media

Before you use the options in the **Archive** menu, make sure that the disk or tape that you intend to use is properly inserted. If you wish to write to a disk or tape, check that the physical mechanism used to enable writing is set on the disk or tape. For 5¼ inch disks, you must remove the write protect tab; for 3½ inch disks and quarter-inch cartridge tapes, move the slider or turn the wheel to the position that allows writing. If you need more information about using disks or tapes, ask your system administrator.

You must also make sure you know the device address for the tape or disk drive you intend to use. (The device address is the name that the system knows the disk or tape drive by.) For information on device addresses and how to set yours, see "Specifying the archive device address" (page 40).

After inserting your disk or tape and specifying the device address, you need to choose a type of format for archiving. The three available types of format are **tar**, **cpio**, and DOS. Generally, **tar** is the most useful format. If your system does not support the use of **tar**, use **cpio**. You can use the DOS format to copy files to a single disk, but it does not support archiving of whole directories, nor does it support tape formatting. To carry out your choice of archive format, see "Specifying the type of format for archiving" (page 39). Once you have chosen an archive format, use that format for all future archiving so that your disks or tapes are interchangeable.

The operations that you can perform with the **Archive** options are discussed in the following sections.

Formatting a disk or tape

Before archiving files, you might have to format the disk or tape that you want to archive them on. First make sure you have set your format type, as described in "Specifying the type of format for archiving" (page 39). If you are not sure what format type is used on your system, or if you are not sure whether you need to format your disk or tape, check with your system administrator. If you do need to format the disk or tape and the type is set correctly, choose the **Format** option on the **Archive** menu. The Media Formatting form appears.

```
┌─────────── Media Formatting ───────────┐
│                                         │
│  Device:     [/dev/rfd096ds15        ]  │
│                                         │
│  Confirm:    ▐Continue▌    Abort        │
│                                         │
└─────────────────────────────────────────┘
```

To format a disk or tape on an archiving device other than the default one, change the entry in the "Device" field; press ⟨F3⟩ for a list of available devices. Otherwise, press ⟨Enter⟩ to accept the default device. For more information on this choice, see "Specifying the archive device address" (page 40).

To confirm your decision to begin formatting, press ⟨Enter⟩ again. Formatting takes a few minutes. You are then returned to the **Manager** menu.

If you decide not format the media, select the **Abort** option in the "Confirm" field.

It is important that you make sure that you format enough media to archive your data. If you are unsure of how much media is required, ask your system administrator.

Copying files to disk or tape

The **Create** option copies files from your current directory to disk or tape. When you select the **Create** option, the screen displays a list of the files in your current directory. Select the file(s) that you want to copy. (If you are making a DOS archive, each file must be individually selected.) If you are using **tar** or **cpio** and you select a directory, all of its contents, including any subdirectories and their contents, are transferred onto the disk or tape.

Press ⟨Enter⟩ to begin the copying process. If the archive fills more than one volume (disk or tape), a prompt to insert a second volume is displayed. After the files that you selected have been copied to the archive, you automatically return to the **Manager** menu. (DOS archive does not create multivolume archives.)

When you make a **tar** or **cpio** archive, all existing files on the archive media are overwritten. When you make a DOS archive, only those files on the archive that have the same name as the files you are copying are overwritten; all others remain intact. Remember that you can archive files but not directories in the DOS format.

Listing files on disk or tape

Before copying files from a disk or tape to your current directory, you might want to see what files it holds. Select the **Archive** menu's **List** option to display a list of the files on the tape or disk. If the list fills more than one screen, use the movement keys to scroll through it. If you see a prompt for a second or subsequent volume, insert the next disk or tape of the archive you are listing. Make a note of the files that you want to retrieve. When you finish looking at the list, press ⟨Esc⟩ to clear the screen and return to the **Manager** menu.

Extracting files from disk or tape

The **Extract** option in the **Archive** menu copies files from disk or tape into your current directory.

When you choose the **Extract** option the Media Extraction form appears.

```
┌──────────── Media Extraction ────────────┐
│                                           │
│   Device:     [/dev/rfd096ds15         ]  │
│                                           │
│   Extract:    █All Files█    Selected Files │
│                                           │
└───────────────────────────────────────────┘
```

To extract files from a disk or tape drive other than the default one, change the entry in the "Device" field. Otherwise, press ⟨Enter⟩ to accept the default device. For more information on this choice, see "Specifying the archive device address" (page 40).

To copy all files in the archive to the current directory, leave the **All Files** option selected for the "Extract" field and press ⟨Enter⟩. Copying begins immediately. If more than one volume (one tape or disk) is required for the archive, a prompt to insert a second volume is displayed. When archiving is finished, you automatically return to the **Manager** menu.

If you want to copy only certain files, select the **Selected Files** option of the "Extract" field. In response, a list of all files on the disk or tape appears on the screen. The list looks similar to the following example. If you are listing DOS files, the filenames appear in all uppercase characters.

```
                                                                    Extract

Select the files to be extracted.
/u/perry/mailfolders                      Wednesday February 22, 1995  4:07

    ../            dead.letter    lenore.notes    review
    mailfolders/   id.spec        mymbox          review.bak
    Outgoing
                   ─────────── Media Extraction ───────────
                   Device:     [/dev/rfd096ds15            ]

                   Extract:    All Files    Selected Files
```

Use the movement keys to scroll through the list. Select the files you want to extract with the ⟨Space⟩ bar, and press ⟨Enter⟩ to copy the files into your current directory.

Specifying the type of format for archiving

The **Type** option in the **Archive** menu specifies the format you intend to use for archiving. When you select the **Type** option, the following form appears:

```
        ─────────── Archive Format ───────────

        Type of Format:     tar   cpio   DOS
```

Select the archive file format that is best suited to the archiving you intend to do. In most cases, **tar** format is the best choice. See your system administrator if you are not sure which format to select.

Specifying the archive device address

The **Device** option in the **Archive** menu specifies the address of the device driver for the device you intend to use for archiving. When you select the **Device** option, the following form appears:

```
┌───────────── Default Media Device ─────────────┐
│                                                 │
│  Device:      [▮dev/rfd096ds1▮              ]    │
│                                                 │
└─────────────────────────────────────────────────┘
```

The "Device" field automatically displays the address of the drive that is normally used for archiving. If you intend to use that drive, just press ⟨Enter⟩ to accept this entry.

To see a list of all possible drives on your computer system, press ⟨F3⟩. A box appears on the screen with a list of the devices in the */dev* directory on a typical system; your list might have additional or different entries.

```
┌───────────── Default Media Device ─────────────┐
│                                                 │
│  Device:      [/dev/rfd096ds1▮              ]    │
│              ┌─────────────────────────┐        │
└──────────────│ *  /dev/rfd048ds9       │:───────┘
               │    /dev/rfd148ds9       │
               │    /dev/rfd096ds15      │
               │    /dev/rfd196ds15      │
               │    /dev/rfd0135ds9      │
               │    /dev/rfd1135ds9      │
               │    /dev/rfd0135ds18     │
               │    /dev/rfd1135ds18   v │
               └─────────────────────────┘
```

The names of the devices appear in coded form. According to the conventions of the code, the first letter of the filename (*r* in the sample list) stands for the access method; you can usually ignore this letter, as it is the same for most devices.

The next two letters specify the type of media; *fd* stands for floppy disk and *ct* for cartridge tape. The next number (0 for most devices in the sample list) is the number of the disk or tape drive. The final characters of the code (for example, *48ds9*) are size parameters for floppy disks. They indicate the number of tracks per inch, the number of sides used (ss=single-sided, ds=double-sided), and the number of sectors per track.

The following table indicates the size and type of media associated with each drive in the sample list.

Device code	Size and type of disk or tape
rfd048ds9	5¼ inch 360KB floppy disk
rfd096ds15	5½ inch 1.2MB floppy disk
rfd0135ds9	3¼ inch 720KB micro-floppy disk
rfd0135ds18	3½ inch 1.44MB micro-floppy disk
rct0	standard tape drive
rctmini	mini-cartridge tape drive

Choose a drive from this list by highlighting it with the arrow keys and pressing ⟨Enter⟩.

The */etc/default/tar* file on your system must contain valid archive devices for your choice to take effect. If you do not know what devices your system supports, or if you do not know which device to choose, see your system administrator.

Using the clipboard from the Manager menu

The clipboard is a temporary holding area that you can use to transfer information between applications. It is used by SCO Shell, the electronic mail application, and the calendar. For example, you can copy your engagements for the current month from the calendar to the clipboard, and paste the clipboard file into some mail to a colleague.

The **Transfer** option on the **Manager** menu allows you to access the clipboard from SCO Shell. Use clipboard to transfer information between different applications. Once you have put an item on the clipboard, you can access it from any application that contains a **Transfer** menu. The **Transfer** menu allows you to copy files to the clipboard that did not come from another application.

To use the clipboard, select Transfer from the **Manager** menu. A menu like the following appears:

```
                                                          Transfer
Copy Paste Remove Quit
Copy file(s) to the clipboard
/u/perry/mailfolders              Wednesday February 22, 1995  4:40
```

The four options on the **Transfer** menu allow you to use the clipboard from the **Manager** menu. Below is a summary of these options:

Option	Description
Copy	copies an item to the clipboard
Paste	pastes an item from the clipboard to a directory
Remove	deletes an item from the clipboard
Quit	exits the **Transfer** menu

Setting preferences for text editing

The **Preferences** option lets you customize how you edit text files. When you select the **Preferences** option, the Editor Configuration form appears.

Use the "Line Length" field to set the default right margin for all use of the built-in SCO Shell text editor. The default is set to the maximum for your screen width; in most cases, this is is 77 spaces. To create a wider right margin as you edit text files with the SCO Shell editor, enter a smaller number in the "Line Length" field. For example, to create a right margin of 10 spaces where your default line length is 77, change the number in the "Line Length" field from 77 to 67.

Use the "Auto Editor" field to specify the pathname of an editing program other than the built-in SCO Shell text editor. For example, enter the path */usr/bin/word* to use Microsoft Word if your system has Word in this location. If you do not know the location of your chosen editor, see your system administrator.

In the "File Suffix" field, specify a filename extension used by your chosen Auto Editor. For example, you can enter **dcx** if you have specified Microsoft Word as your Auto Editor. After you have set these preferences, select the **Edit** option in the **Manager** menu, then select any Word file with the *.dcx* extension, and you see the text in a Microsoft Word editing window. Files that do not have the *.dcx* extension still invoke the SCO Shell text editor.

If you specify an Auto Editor but leave the "File Suffix" field blank, SCO Shell invokes your Auto Editor for all files you select after choosing the **Edit** option in the **Manager** menu.

The last field in the form allows you to save your preferences for future SCO Shell sessions.

If you make the changes described above, the form looks like the following:

```
┌────────────────────Editor Configuration─────────────────────┐
│ Built-in text editor                                         │
│                                                              │
│      Line Length: [67 ]                                      │
│ User selected editor                                         │
│                                                              │
│      Auto Editor:   [/usr/bin/word                      ]    │
│      File Suffix:   [dcx             ]                        │
│                                                              │
│ Save as default?   [Yes]    No                               │
│                                                              │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Exiting the Manager menu

Select the **Quit** option to exit the **Manager** menu. You return to the SCO Shell window.

Running utilities and applications

SCO Shell provides two top level menu items to enable you to run other programs; **Utility** and **Application**. In general, a utility is a small program that provides you with some useful information about the computer or allows you to do a particular task. For example, a utility might tell you who else is logged on, or where a file with a given name is stored. Applications are larger programs (such as word processors, spreadsheets or databases) that you interact with in depth. SCO Shell comes with three applications; the electronic mail program described in the *Mail and Messaging Guide*, a calculator (see "Using the Calculator" (page 72)) , and a calendar (see "Using the Calendar" (page 49) for details).

For an explanation of how to run a utility or application, see "What utilities are available" (page 44) and "What applications are available" (page 47) respectively. (Alternatively, select the **Application** or **Utility** main menu items, select the program you want to run, and press ⟨Enter⟩.)

You can issue commands from SCO Shell if you are familiar with this way of running programs. Type an exclamation mark (!) then enter your command and press ⟨Enter⟩. When the program has completed, SCO Shell will resume.

When you run an application or utility, you interact with the program instead of SCO Shell. The SCO Shell waits until the program you have invoked finishes before accepting any more commands; consequently, if you run an application that does not follow the standard SCO Shell keystrokes (such as a word processor from another company, or the text editor **vi**), you may find yourself in unfamiliar territory until you quit the application. A golden rule, that you should follow before running an unfamiliar application, is to look up how to leave the program and make a note of it. Otherwise you may have difficulty returning to the SCO Shell.

What utilities are available

The utilities that SCO Shell recognizes are listed in the Utilities window. To see the Utilities window, select **Utility** and press ⟨Enter⟩. The menu bar vanishes, and a list of utilities appears in a window on the screen. The top item on the list is highlighted; if you press ⟨Enter⟩ again, that utility will be activated.

It is possible to install many utilities on your system; not all of them will fit in the window at the same time. You can tell that there are items outside the visible area if the right hand edge of the window is showing a scroll bar (a slider control that you can drag up and down or move using the arrow keys).

If you use the up and down arrow keys, the list of available utilities moves up and down behind the window to reveal the additional items:

```
┌─────────────────────────────────────┐
│* List Users    - Who is Logged on  :│
│  Compare       - Compare Files      │
│  Find          - Locate A File      │
│  Search        - Text Search        │
│  Usage         - Disk Usage         │
│  Disk Free     - Space Available    │
│  Kill          - Kill A Program     │
│  Processes     - List My Processes  v│
└─────────────────────────────────────┘
```
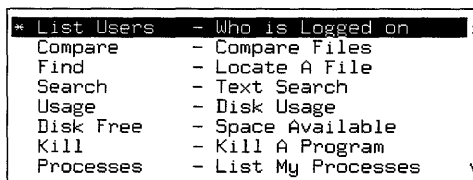
If you do not want to run a utility, you can return to the menu by pressing ⟨Esc⟩.

A number of utilities come with SCO Shell. Here is a brief explanation of what they do:

List Users   Lists all the users who are currently logged into the system, along with their terminal and the date and time at which they logged in.

Compare   Compares two files, visually highlighting the differences between them. You are prompted for the type of the file (whether a binary (program-like file), or a text file, or a directory), then to select two filenames. **Compare** then presents you with a visual indication of the lines or characters that differ in the second file with respect to the first.

Find   Locates a file in the filesystem. You are prompted for the name of a file to look for. **Find** then searches for everything that matches this filename.

Search   Searches for text contained in a given file. You are prompted for a piece of text to look for, then for a file to search. **Search** reports if it locates the text in question.

Usage   Reports on how much space is used by the files stored below the current directory. The output is reported in disk blocks, and is broken down by directory.

Disk Free   The opposite of **Usage**; reports how much space remains available on the filesystems currently accessible to your machine.

Kill   Allows you to terminate a program that you are currently running. For an explanation of this feature, see Chapter 5, "Controlling processes" (page 157).

Processes   Lists the currently running processes under your control. (The SCO OpenServer system allows you to run several programs simultaneously; a running program is termed a "process".)

All procs   Lists all the processes currently running on the computer. (There will be a large number of these, and you will probably be unable to kill any that do not belong to you. In general, do not attempt to destroy processes unless you know what they are for; if you succeed, you may impair the functioning of the system.)

OS Version   Prints a message containing technical information about the type of system you are running on the computer.

Set colors   If you are working on a color terminal or console, this tool allows you to select the colors in the applications and utilities that are displayed. You must pick an application from the list that you are presented with, then select the particular object within that application to change color; the utility then allows you to pick new colors for that object. You should save your settings when you finish using this utility. The next time you start the changed program, its colors will have changed.

Adding a utility to the list

To add a utility to the list, from the top level menu, select:

Options ⇨ **Utillist**

The **Utillist** menu allows you to add a command to the list, edit or delete an item on the list, or undelete an item you have deleted by mistake.

You can also create a folder; that is a menu item containing other items. (When you select the **Create** menu option within the **Add** menu you are asked whether you want to create a command that can be added to your list, or a folder.) Folders are displayed on the list with trailing ellipses (...), and from the **Utillist** menu you can open a folder and add commands within it. Thus, by using folders you can define multiple levels of nested utility commands.

When you edit a command or create a command, you are given a form to fill in that contains fields for the command name, a brief description (displayed on the line below the menu when the command is highlighted) and the path-name of the command (the absolute pathname required to execute it):

```
┌────────────────────Creating command─────────────────────┐
│  Name:          [              ]                         │
│  Description:   [                        ]               │
│  Path Name:     [                                 ]      │
└──────────────────────────────────────────────────────────┘
```

For an explanation of pathnames, see "How directories are organized" (page 84).

Once you have created a new command and added it to the list or to a folder that is added to the list, you can run it by selecting:

Utility ⇨ *command_name*

What applications are available

Applications are larger programs that usually require you to interact with them extensively. (Utilities, in contrast, usually do just one thing and require little interaction.)

The standard applications that are available to SCO Shell are the e-mail program, the calendar, and the calculator. For details of these programs, see the *Mail and Messaging Guide*, "Using the Calendar" (page 49) and "Using the Calculator" (page 72) respectively.

You run an application the same way you run a utility; the only difference is that you select the application from the **Application** item rather than the **Utility** item on the top level menu.

Adding an application to the list

To add an application to the SCO Shell menu, select:

> **Options** ⇨ **Applist**

The procedure is essentially the same as for adding a utility (see "Adding a utility to the list" (page 46)), except that the application appears on the application list instead of the utility list.

Copying items between applications with the clipboard

Different applications treat items stored on the clipboard in different ways; for example, the electronic mail application may see an item as an "attachment" (a piece of data appended to a mail message) while the SCO Shell sees it as a file.

To copy an item to the clipboard, select:

> **Manager** ⇨ **Transfer** ⇨ **Copy**

You are prompted for an item to place on the clipboard. SCO Shell is able to place files on the clipboard; applications may be able to place other types of information on it.

To paste an item on the clipboard into a file, select:

> **Manager** ⇨ **Transfer** ⇨ **Paste**

You are prompted for a named item on the clipboard, and the name of the file in which to place it. Note that if you give **Paste** a filename that already exists, the file will be overwritten with a new one containing only the clipboard item you are saving.

Items placed on the clipboard are referred to by name and stay there until you explicitly remove them. To remove an item from the clipboard, select:

Manager ⇨ **Transfer** ⇨ **Remove**

The SCO Shell prompts you for the item to remove.

Printing files

Select the file to be printed from those listed using:

Print ⇨ **Go**

The procedure for printing several files from within SCO Shell is the same as for printing a single file, except that instead of only selecting one file, you use the spacebar to select several files.

When you select the menu item to print a file, the file is sent to the print queue. A print queue is a queue of files waiting to be printed on a specific printer. Because an SCO OpenServer system may have many users, any or all of whom may be printing files, your file may not be printed at once; it is *spooled* to the back of the queue. If it is the only file waiting to be printed, it is processed at once.

Once you have sent a job to the printer you can carry on with another task; you do not have to wait for the print job to commence or complete.

Displaying or canceling print jobs

You may wish to find out the status of print jobs that you submitted at an earlier time (you may need this information to cancel a print job). Select:

Print ⇨ **PrintStatus**

to display the list of currently queued print jobs.

You can only cancel print jobs that you requested. Select a print job to be canceled from those listed using:

Print ⇨ **Cancel**

Selecting a printer

If you know that several printers are connected to your system, and you want to send a file to a printer that is not busy or that has special printing capabilities, you need to know the destination printer's name.

Select a printer from those listed using:

Print ⇨ **Select**

Chapter 2

SCO Shell accessories

Chapter 1, "Using SCO Shell" (page 11) explains how to use SCO Shell to manage your files and directories, and run other utilities. This chapter explains how to use two more applications available to you through SCO Shell:

- Calendar (this page)

- Calculator (page 72)

Using the Calendar

The SCO Shell Calendar lets you organize your daily schedule. It also provides access to the calendars of other users, so you can add events to their calendars as well as to your own, checking their calendars quickly for free times or scheduling conflicts. If your computer is on a local network, you may even have direct access to the calendars of users who work on other computers on the network.

In addition to your personal calendar, you can create calendars for facilities, equipment, or any other resource. For example, you can create a calendar for a conference room so that anyone who wants to schedule the room for a meeting makes an entry on that calendar for the appropriate time. Other people, checking the room's calendar to schedule their own meetings, can then see if the room is already booked when they want to use it.

You have control over who can access your personal calendar, and how much they can change it. You can set up your calendar so that other users can change and delete the events that are already on it, or you can restrict them to adding new events.

If you want even more privacy, you can give them view permission, which only allows them to look at your events, or blind view permission, which allows them to see the times you have scheduled but no details about your events.

For complete privacy, you can specify yourself as the only person allowed to view your calendar. Because you can give different permissions to different users or groups of users, you can easily strike a balance between security and convenience.

Starting the Calendar

To run the Calendar, select:

Application ⇨ **Calendar**

The opening screen is displayed showing the day's events. It looks something like:

```
                                                              Calendar
Next  Back Goto Add Change Delete View Print Transfer Options Quit
Move to the next day
Calendar: perry                              Wednesday February 22, 1995   2:25
                         Thursday February 23, 1995
     8:00- 9:00    Performance review    Alice's office
     9:00-10:00    Staff meeting         Boardroom
    10:00
    11:00
    12:00- 1:00    Lunch                 Trattoria
     1:00
     2:00
     3:00- 3:30    Sales meeting         Downstairs Conf room
     4:00
     5:00
```

If you want to see more information on each event, press ⟨F6⟩. The display then expands to show the details of each event, plus a list of the people who are scheduled to attend. To make all this extra information disappear from your screen, press ⟨F6⟩ again.

You can set your calendar display to show this additional information by default. For details, see "Setting Calendar options" (page 65).

Quitting from the Calendar

To exit the Calendar, select **Quit** from the **Calendar** menu bar. You can also bring up the **Quit** menu item by pressing ⟨F2⟩ from anywhere in the Calendar program.

Moving between days

When you start the Calendar, SCO Shell displays the current day's events by default.

To display the events for the next day, select **Next** from the **Calendar** menu item. With the **Next** item still highlighted, you can continue pressing ⟨Enter⟩ to move the calendar forward one day at a time. You can go back by selecting the **Back** item.

Scheduling events

The Calendar allows you to schedule events on any date between 1900 and 2099. Select **Goto** and a three-month calendar appears at the bottom of the screen with the current month in the middle.

The cursor is positioned over the current date on the three-month calendar. Use the arrow keys to move the date highlight. The ⟨PgDn⟩ and ⟨PgUp⟩ keys move the cursor to the same date in the next and previous months, respectively. The ⟨Home⟩ key returns you to the current date.

Instead of using the cursor to select a new date, you can enter a date from the keyboard. When you enter a date at the prompt, the Calendar automatically assumes the current year and month unless they are specified. To specify the 30th of this month, type **30** at the prompt and press ⟨Enter⟩. Similarly, **11/4** is an acceptable way of specifying November 4 of the current year.

You can specify a year either by typing out the whole year (as in **1992**) or by giving only the last two digits (as in **92**). You cannot specify a date that is more than 50 years ago using only two digits for the year. For example, the Calendar interprets **5/5/33** as May 5, 2033, not 1933. You would have to enter **5/5/1933** if you meant the year 1933.

Here are some sample dates and the Calendar's interpretations of them:

Example	Interpreted as
6	sixth day of the current month
7/4	July 4 of the current year
12/7/52	December 7, 1952
10/3/29	October 3, 2029
Nov 11	November 11 of the display year
1 april	April 1 of the current year
+5	five days past the current date
+5 w	five weekdays past the current date

You can make special uses of the " + " or " - " characters if you are filling the "Start Date" and "End Date" fields in the Repeat form for a repeating event. See "Scheduling repeating events" (page 55) for more information.

If you enter an invalid date format, SCO Shell gives you an error message telling you so. Enter a valid date and press ⟨Enter⟩.

After selecting the new date

Once you select a new date, either by entering the date from the keyboard or by pressing ⟨Enter⟩ on the highlighted date, the screen displays the events for the new date. The three-month calendar is no longer displayed on the screen. You can now use the **Add, Change**, and **Delete** functions on the currently displayed date's events.

Returning to the current date

You can return quickly to the current date from the **Calendar** menu by selecting the **Goto** option then pressing ⟨Enter⟩.

Scheduling a meeting or event

You can schedule events to occur either once, or repeatedly at regular intervals.

Scheduling single events

To schedule an event on the Calendar:

1. Go to the day on which you want the event to happen.

2. Select **Add** ⇨ **Event** from the **Calendar** menu.

```
                                                              Event
Select a time or <F10> to add a timeless event
Calendar: perry                      Wednesday February 22, 1995   2:26
                      Thursday February 23, 1995
*   8:00
    9:00
   10:00
   11:00
   12:00
    1:00
    2:00
    3:00
    4:00
    5:00
```

3. Pick the time at which you want the event to take place using the arrow keys, and press ⟨Enter⟩.

4. If you want to select a time slot which is not displayed, press ⟨F10⟩. This allows you to enter a timeless event or specify a start time other than those displayed. The following is an example of a timeless event form:

```
                        Thursday February 23, 1995
                          ─────── Event ───────
        Date [Feb 23 1995        ]
        Time [           ]       Created  Feb 23 1995  10:48 a.m.
    Duration [           ]       Caller   perry
        What [                   ]
       Where [                   ]
     Details [                                                      ]
             [                                                      ]
         Who [                                                      ]
             [                                                      ]
             [Public]  Private
```

You can now enter information about the event you are creating.

The Time and Duration fields

When you enter times between 8:00 and 11:59 in the "Time" field, the time defaults to a.m. The times between 12:00 and 7:59 default to p.m. You can enter **am** or **pm** after the time to override the default.

Enter whole numbers (1, 2, 3) in the "Duration" field to signify hours. If the duration includes increments of less than an hour, enter the time in hours and minutes, or hours plus a decimal fraction. For example, an event that lasts two hours and 15 minutes is entered in the "Duration" field as **2:15** or **2.25**.

If you enter a number below 10, the duration is interpreted as hours. A duration of 10 or greater is interpreted as minutes. You can schedule an event for less than 10 minutes by preceding the number with a colon, for example, **:08** minutes.

Here are some sample times and durations and the Calendar's interpretations of them:

Example	Interpreted as
8	a time of 8:00 a.m.
7	a time of 7:00 p.m.
:11	a duration of 11 minutes
11:	a duration of 11 hours
11	a duration of 11 minutes
3:15	a duration of three hours and 15 minutes

The Caller field

The "Caller" field displays one name on your own calendar, or two names if the current calendar is someone else's. For example, the "Caller" field displays eric (jim), if *jim* has logged in and then switched to *eric's* calendar. This field is set automatically and cannot be changed. For more information on switching to another user's calendar, see "Accessing other calendars" (page 65).

When the event is posted, it automatically shows up on *eric's* calendar. It only appears on *jim's* calendar if *jim* is specified in the "Who" field of the Add form. By default, users' logins are also their calendar names. To make a calendar under another name, see "Creating a new calendar" (page 66).

The Who field

You may enter only existing calendar names in the "Who" field of the Add form. Acceptable calendar names are login names, names created as described in the "Creating a new calendar" (page 66) section of this chapter, and aliases (see "Managing aliases" (page 70) for further information about aliases).

To display a point-and-pick list of all current calendar names, press ⟨F3⟩ while the cursor is in the "Who" field. You can search the list for a login by pressing ⟨F5⟩ and keying it in.

When you finish entering information in the form, press ⟨Enter⟩ or ⟨Ctrl⟩X to update the calendar.

If you enter a calendar name that does not exist in the "Who" field of the Add form, SCO Shell gives you the message:

> *Wrong name*: No such calendar

but it does not delete the name from the list. To correct a name or delete an outdated name from "Who", see "Changing an event" (page 60).

If there is a scheduling conflict involving any of the invitees for the new event, a box pops up on the screen with the invitee's name and a description of the conflicting event. If the conflict is acceptable, select "Yes"; you return to the **Calendar** menu. The person with the conflict is invited, and the time slot in question now contains more than one event in their calendar.

If the conflict is not acceptable, select "No". You then return to the Add form so that you can reschedule the event.

If you enter the name of a calendar for which you do not have schedule permission, the Calendar displays an error message in the bottom-left corner of the screen. For more information on permissions, see "Setting permissions on the current calendar" (page 68).

Scheduling repeating events

Use the **Repeat** function to schedule events in your calendar that occur on a regular basis. For example, if your weekly staff meeting occurs at 2:30 p.m. each Monday, use the **Repeat** function to schedule the meeting as a recurring event.

Select a time slot by selecting the desired line and pressing ⟨Enter⟩, or by typing the first number of the selected time. If you want to schedule an event for a time in between those shown, press ⟨F10⟩, and the Repeat form appears with the "Time" field blank for you to enter a time:

```
┌──────────────────────────── Repeat ──────────────────────────────┐
│ [Weekly]  Continuous    Periodic    Biweekly    Date      Relative    Yearly │
│                                                 Monthly   Monthly            │
│           Sun Mon Tue Wed Thu Fri Sat                                        │
│           [ ] [ ] [*] [ ] [ ] [ ] [ ]           Period                       │
│                                                                              │
│  Start Date [Apr 25 1995      ]    End date [                    ]           │
│      Time [           ]        Created   Apr 25 1995 10:51 a.m.              │
│  Duration [           ]        Caller   perry                                │
│      What [                    ]                                             │
│     Where [                    ]                                             │
│   Details [                                                      ]           │
│           [                                                      ]           │
│       Who [  ·                                                   ]           │
│           [                                                      ]           │
│           [Public]   Private                                                 │
└──────────────────────────────────────────────────────────────────┘
```

Enter the date on which you wish to start the repeating event in the "Start Date" field. The current date is the default. Entering **+3w** in the "Start Date" field advances it by three weekdays. When you use the "w" suffix, the Calendar program counts weekdays only, and adds or subtracts them relative to the display date. For instance, if the original starting date is Monday, October 18, and you enter **-1w**, the new starting date is Friday, October 15.

Enter the last date on which you wish to schedule the repeating event in the "End Date" field. The Calendar adds or subtracts days relative to the starting date. For example, if the starting date is Friday, October 15, and you enter **+5w** in the "End Date" field, then the ending date becomes Friday, October 22.

Selecting the type of repeating event

At the top of the Repeat form are options for how frequently you want the event to occur.

You can use the ⟨Home⟩ key or ⟨Ctrl⟩**T** to move to the top of the form. Then, to choose the item that you want, either use the ⟨Space⟩ bar or arrow keys to move the highlight between the options, from **Weekly** through **Yearly**, and press ⟨Enter⟩, or type the first character of the desired option:

Weekly or Biweekly	Select the day(s) of the week from the line below on which you wish to schedule an event. Press ⟨Space⟩ to toggle the fields on or off. The * character indicates that a field is toggled on. The day of the week of the starting date of the event is toggled on by default.
Continuous	Schedules the event to occur every day between the starting and ending dates.
Periodic	Schedules the event to occur at intervals of a number of days (between 2 and 255). When you enter the number of days in the cycle in the "Period" field, keep in mind that the Calendar counts weekends and holidays.
Relative	Schedules the event on the same day relative to some repeating unit of time. These include the Date (so the event recurs on the same date of the month), the Month (so the event occurs on the same day in the month, for example, the second Tuesday of every month), and Yearly (so the event recurs on the day every year).

Selecting Public or Private access

Select **Public** to allow all users to see the information in the "Who," "What," "Where," and "Details" fields for the event that you are scheduling. Select **Private** to allow only you (as the owner of the event) and invitees (specified in the "Who" field) to have access to the information in these fields (either through mail or through their own calendar).

Notifying invitees about an event

If the repeating event that you are adding is a scheduled meeting with other users, the information for the event is automatically updated on each of the calendars for which you have schedule permission. Mail is sent to each of the invitees if they have the "Mail notification" field in their **Options** ⇨ **Configure** forms set to "Yes". See "Setting permissions on the current calendar" (page 68) for more information about this form.

If you try to schedule an event with a calendar for which you do not have schedule permission, a message appears telling you that you do not have schedule permission and that mail is being sent instead. The mail message invites the owner of the calendar to the event. This mail notification is sent only for the first occurrence of the event. It is up to the invitee to remember all future occurrences.

If the Calendar is not associated with a login, mail is not sent. See "Creating a new calendar" (page 66) later in this chapter for more information.

If the event is cancelled after its scheduled starting time, mail regarding the cancellation is not sent to the invitees.

Changing or deleting a repeating event

When changing or deleting a repeating event, you can choose to change all occurrences of the event, or just one instance of it. For example, if you have scheduled a board meeting for every Wednesday at 3:00, you can change the time to 2:00 for one week's meeting without affecting the times of the other meetings.

Changing a repeating event

Go to the date on which the event is to take place, and select **Change**. You can select **Every** to change all instances of the repeating event, or **Today's** to change only the display date's occurrence of the repeating event. For details on changing the information about an event, see "Changing an event" (page 60).

Deleting a repeating event

Select **Delete** to delete one or all occurrences of a repeating event. When you select a repeating event, a **Delete** menu appears.

If you do not have permission to change the event, you see "Yes" and "No" options. Choose "Yes" to delete the event from your own calendar, and choose "No" to leave it on your calendar.

If you have permission to change the event (either you scheduled it, or you have full permission on the owner's calendar) you see a menu with three options: "No", "Today's", and "Every". Make your selection and press ⟨Enter⟩.

If you had invited other users to the event, the changes made to the event will show up on their calendars when they next access them.

For details on removing a non-repeating event, see "Deleting an event" (page 60).

Searching for free time

If you are scheduling an event with several invitees and you are unsure of their free time, select **View** ⇨ **Free** and specify their names in the "Who" field on the form displayed. You can use the information displayed to schedule an event that all invitees are able to attend. See "Viewing a free-time list" (page 63) for more details.

Alternatively, if you need to search for a free time over a long time period, select **Add** ⇨ **Search** to display a Search form.

Change any of the default entries you need to, and specify the logins of those people who *must* attend the event in the "Who" field, and those whose attendance is voluntary in the "Optional" field. By pressing ⟨F3⟩ you can access a point-and-pick list of calendar names. You can search the list for a specified login by pressing ⟨F5⟩.

When you have entered all the necessary information, press ⟨Enter⟩ from the last field or ⟨Ctrl⟩**X** to begin the search. The Calendar program then searches the calendars of all the people listed in the "Who" and "Optional" fields of the Add form for times when all of those users are free. Note that if a person is listed in the "Optional" field, they may be invited to attend even if they have conflicting events.

Choosing a free time

If one or more common free times are found, the free-time list is displayed over the right half of the screen with the first available date and time combination highlighted in the "Free Times" window. To schedule the event, move the highlight to the desired date and time in the "Free Times" window and press ⟨Enter⟩. This exits you from the list and takes you to the Add form. The "Date," "Time," and "Duration" fields of the Add form are automatically filled in with the information from the Search form. Fill in the remaining fields on the Add form as explained in "Scheduling a meeting or event" (page 52).

You can change the search parameters while you are viewing the list of free times. To do this, press ⟨Esc⟩ to return to the Search form. Edit the form to change the parameters of the search. Press ⟨Enter⟩ in the last field (or ⟨Ctrl⟩**X** anywhere on the form) to start the new search for common free times.

Resolving scheduling conflicts

If the event that you are adding conflicts with a previously scheduled event in either your calendar or the calendar of an invitee, the calendar automatically displays the conflicts at the bottom of the screen and asks whether they are acceptable.

If the conflicts are unacceptable, either select "No" in the last field and press ⟨Enter⟩, or type **n** to return to the Add form. Edit the "Date," "Time," and "Duration" fields to resolve the scheduling conflict. If the conflicts are acceptable, select "Yes". The scheduling conflicts are ignored, and the current event is scheduled in the calendars of all the people listed in the "Who" field in addition to any event previously scheduled.

Scheduling timeless events

You can use the Calendar to schedule an event that does not occur at a specific time. For example, you can schedule a task as an event that you need to complete on a specific day, but not at a specific time. A timeless entry appears as the first scheduled event for the day, marked with a dash " - " character.

To schedule a timeless entry, clear the "Time" field in the Add form. Complete the form with the desired date and either press ⟨Ctrl⟩X or press ⟨Enter⟩ on the last field to schedule the event.

As a shortcut for entering a timeless event, press ⟨F10⟩ when selecting a time slot for the event on the calendar display. The Add form appears with the "Time" field already blank.

To schedule a repeating event without a specific time, see "Adding "To do" items to the Calendar" (this page).

Scheduling durationless events

Sometimes you know when an event will take place, but not how long it will last. In such cases, you can schedule an event that has no known duration. To do this, either enter a 0 in the "Duration" field of the Add (or Repeat) form, or leave it empty. Complete the form according to the instructions in "Scheduling a meeting or event" (page 52).

Adding "To do" items to the Calendar

You can use the Calendar as a "To do" list by scheduling events that have no set times but repeat every day. The "To do" entries appear as the first scheduled events for the selected day only and are marked with dash characters. These entries appear on the current day's calendar every day until you remove them.

To schedule a "To do" item, select **Add** ⇨ **Event** to add a single entry, or **Add** ⇨ **Repeat** to add a repeating entry to your calendar. Press ⟨F10⟩ to select a timeless event, and fill in the details on the displayed form. (Leave the "Time" and "Duration" fields on the form blank.) For details on how to fill out a Repeat form, see "Scheduling repeating events" (page 55).

You can create a "To do" reminder for a single day by following the procedure described in "Scheduling timeless events" (page 59).

Changing an event

To change the information about a scheduled event, select **Change** from the **Calendar** menu. If the event is scheduled for another day, first **Goto** the correct date.

A scheduled event can be changed by the person who originally created or scheduled the event, or by anyone with full access permission on the owner's calendar. Write, view, and full access permissions, and how to set and change them, are discussed in "Setting permissions on the current calendar" (page 68).

Deleting an event

To delete an event from your calendar and the calendars of other users, select **Delete** from the **Calendar** menu. The first scheduled event for the current date is highlighted.

Select the entry you want to delete by choosing the desired line and pressing ⟨Enter⟩.

This deletes the event from the calendars of the people you invited, provided you originally scheduled the event. If a user who was invited for the canceled event has "Invitation" and "Cancellation" set to "Yes" on their Configure form, you are prompted for up to two lines of text describing the reason for the cancellation. This text is then included in the mail message that is sent to them, notifying them of the change in their calendars. For more information about the Configure form, see "Setting permissions on the current calendar" (page 68).

An event can be deleted by the the person who scheduled it, or anyone with full access permission on the owner's calendar. Write, view, and full access permissions are discussed in "Setting permissions on the current calendar" (page 68).

You can delete an event from your own calendar even if that event is owned by another person. If you delete a future event, a prompt asks you to state your reason for the deletion. Your response is mailed to the person who scheduled the event. When you delete the event it no longer appears on your calendar; however, if you do not have full access permission on the owner's calendar, the event still exists on the calendars of the other users who are scheduled to attend.

Viewing the Calendar

To display the calendar on your screen, select **View** from the **Calendar** menu. You can use **View** to check a print request before sending it to the printer, or to display the calendar in a specified format.

Event, Week, Month, Range, and **Free** let you choose the format for the displayed calendar.

Viewing an event

Event displays a list of the events scheduled for the current calendar day. If more than one event is scheduled for the current day, select the event you want to view from the list displayed.

Viewing a week

Week displays the events for the week that contains the current day. This display includes the information from the "What" and "Where" fields of the Add form.

```
                                                                    Week

                  Movement keys to scroll   <Esc> to continue
Calendar: perry                            Wednesday February 22, 1995   2:33

                  ┌──────────────────────────────────────────┐
                  │         WEEKLY CALENDAR FOR perry         │
                  │           February 27 - March 5           │
                  ├────────────────────────┬─────────────────┤
                  │               Monday 27 │       Tuesday 28 │
                  │ 1:00- 2:18 Author's meeting │ 1:00- 2:00 Project meeting │
                  │                        │            Boardroom │
                  │                        │                  │
                  │                        │                  │
                  │                        │                  │
                  ├────────────────────────┼─────────────────┤
                  │            Wednesday 1  │      Thursday 2  │
                  │ 10:00-11:00 Publishing Systems │ 9:00-12:00 Training │
                  │                        │            Suite 4 │
                  │                        │                  │
                  │                        │                  │
                  └────────────────────────┴─────────────────┘
```

Viewing a month

Month displays a box calendar for the month containing the current date. Only the information from the "What" field of the Add form is included.

By default, Saturdays and Sundays are combined and put into one box for each week. If you want them to be displayed in separate boxes, follow the instructions in "Setting calendar preferences" (page 67).

If you want to display a different month, select the **Goto** option from the **Calendar** menu.

Viewing a range of dates

Range displays events that occur within a range of dates. The default range is 30 days, starting with the current date.

You may indicate whether you want a brief or detailed description of each event. Select **Brief** to display only the "What" and "Where" fields. Select **Detailed** to display the "What," "Where," and "Details" fields.

In the "Pattern" field, you can specify a pattern of letters or words. If you previously chose "Brief", the Calendar searches the "What" and "Where" fields of each event's Add form for the specified pattern. If you selected **Detailed**, the "Details" field is searched in addition to the "What" and "Where" fields. The Calendar then displays the events that contain the specified patterns and are within the specified date range.

For example, if you want a listing of all board meetings planned for the next three months, set up the appropriate date range and enter **board meeting** in the "Pattern" field.

Note that when SCO Shell searches for a pattern, it does not distinguish between uppercase and lowercase letters, so you can use either type when filling out the "Pattern" field.

Viewing a free-time list

Free graphically displays free times for users specified in the "Who" field. Shaded areas indicate times when users have events scheduled.

```
                                                                    Free
Next Back Add Quit
Choose a time and add an event
Calendar: perry                          Wednesday February 22, 1995   4:40
                           Thursday February 23, 1995

                 8   9   10  11  12  1   2   3   4   5
         perry .   .   .   .   .  ▇▇  .   .   .   .
       charles .   .   .   .   .  ▇▇  .   .   .
        joseph .   .   .   .   .   . ▇▇   .   .   ▄
          kate .   . ▇▇▇▇▇▇  .   .   .  ▇▇  . ▄▇
         sarah .   .   ▇▇▇  .   .   .   .  ▇▇  . ▄
           all .   .  ▇▇▇  . ▇▇▇  ▇▇  . ▄▇
                 8   9   10  11  12  1   2   3   4   5
```

To add a new event to your calendar, select **Add** from the **Free** menu to highlight the first empty slot on the free-time display. Use the ⟨Space⟩ bar to select desired time, and press ⟨Enter⟩ to make your selection.

Printing the calendar

Select **Print** from the **Calendar** menu to print a calendar. The name of the current printer appears at the top of the screen.

Printing an event

Using the arrow keys, select the event(s) that you want to print and press ⟨Space⟩ to mark each event with a " * ". When you have marked all the desired events, press ⟨Enter⟩ to print the information.

Note that with all print commands, the information to be printed is not displayed on the screen. To view the information on the screen, see "Viewing the Calendar" (page 61).

Printing calendars for a day, week, or month

You can select **Day, Week**, or **Month** to print all the events scheduled for the current day, week or month. **Week** omits meeting times, while **Month** prints a box calendar with basic information about each meeting from the "What" field of the Add form.

Selecting an alternative printer

To send the Calendar to a printer other than the configured printer, select **Print** ⇨ **Select** and select the desired printer. Press ⟨Enter⟩ to select it as the current printer for this session. The current printer configuration returns to the default printer the next time you run the Calendar. Choose **Select** from the SCO Shell **Print** menu to change the default SCO Shell printer.

Transferring information from the Calendar to other applications

Transfer on the **Calendar** menu allows you to transfer information from the Calendar to a text file or to other applications using the clipboard.

Select **Transfer** from the **Calendar** menu. The four items on the **Transfer** menu allow you to:

- copy an event, a day, a week, a month, or a range of dates from the Calendar to the clipboard

- paste an event from the clipboard to the current Calendar

- remove one or more items from the clipboard

- quit to the **Calendar** menu

Transferring an event

Select **Transfer** ⇨ **Copy** ⇨ **Event** to copy an event from the current day to the clipboard.

To paste an event into a different day on your calendar, use **Goto** in the **Calendar** menu to pick the day you want, then choose **Transfer** ⇨ **Paste** and select the item you want from the clipboard list that appears.

The Add form appears, with the data for the selected event displayed in the appropriate fields. Edit the data to reflect changes in plans for the event. You can even change the "Date" field if you want to paste the event on a different date from the one displayed. Use ⟨Ctrl⟩X to close the Add form and paste the event into the calendar. If there is a conflict, a form appears, asking you if you want to allow the conflict.

Transferring events for a day, week, or month

Although you can copy the events of a day, week, or month to the clipboard for use in another application, you cannot paste this clipboard entry back into a calendar. Only information about single calendar events can be pasted from the clipboard to a calendar.

You can only transfer events into calendars one at a time, but you can transfer multiple events to text files or to other applications. To copy the events of a day, week, or month, use the procedure described in the previous section for transferring an event, but choose **Day**, **Week**, or **Month** from the **Copy** menu.

If you transfer the events of a day or a week, the data from the "Time," "What," and "Where" fields is copied into the clipboard.

If you transfer the events of a month, only the data from the "What" field is copied into the clipboard.

If you paste any of these clipboard items into an application that displays graphical characters, the calendar shows not only the format but the graphical appearance of a printed copy.

Transferring events from a range of dates

Range on the **Copy** menu transfers events that occur within a range of dates. You can transfer multiple events from the calendar to a file or to other applications.

Setting Calendar options

Using the functions from the **Options** menu, you can:

- access other calendars

- create new calendars

- set calendar-configuration parameters and preferences

- change the blind view, view, schedule, and full-access permissions for the current calendar

- manage aliases

Accessing other calendars

You can access calendars associated with other individuals, particular projects, events, or groups of people, provided that you have view, schedule, or full-access permissions for those calendars. If your computer is on a local network, you may have direct access to the calendars of users who work on other computers on the network.

By default, you can view anyone else's calendar. The owner of a calendar must specifically deny view access to prevent viewing. To access another calendar, select **Options** ➪ **Switch** and enter the name of the calendar to which you wish to switch. You can select from a point-and-pick list of all existing calendar names by pressing ⟨F3⟩, or you can select from a list of login names by pressing ⟨F5⟩.

If you schedule an event while you are switched to this calendar, mail is sent to all the invitees from your login name, not from the calendar to which you are currently switched. However, the event is owned by the current calendar not by you. For example, if user *jane* switches to user *diane*'s calendar and schedules an event, the invitees receive mail from *jane*, not *diane*. However, the event is owned by *diane*.

If you do not have schedule permission, the Calendar program tells you so. You can view the calendar, but you cannot make any changes to it.

If you do not have at least blind view permission on the calendar that you want to switch to, you are given the message that you do not have view permission. If this is the case, you are not allowed to switch to the new calendar. You must press ⟨Enter⟩ to return to the calendar from which you attempted to switch.

For more information on permissions, see "Setting permissions on the current calendar" (page 68).

Creating a new calendar

Make allows you to create new calendars. In addition to calendars associated with users, you can create calendars associated with a project, task, or group of users. Calendars can also be associated with shared resources, such as conference rooms, slide projectors, and other equipment.

When you select **Make**, you are asked for the name of the new calendar. The new name must be no more than 15 characters long without any spaces. (If a calendar by that name already exists, you are given a message to that effect. You must then press ⟨Enter⟩ to return to the **Calendar** menu.)

You can now schedule events with this calendar in the same way that you schedule events with other users. For example, if you need a particular room and a slide projector for your meeting, put the names of the calendars for the room and the projector in the "Who" field of the Repeat or Search form, just as you would for any other invitees. The calendar checks the schedules of these resources to find a common free time. When you select a date and time, the resources, as well as the invitees, are scheduled to attend your meeting.

The calendars created with **Make** are not associated with any particular user. The initial view and schedule permissions are set to allow all users access to a calendar. By default, only the owner of the Calendar is allowed full access. To learn how to change the view, schedule, or full-access permissions, see "Setting permissions on the current calendar" (page 68).

Because these calendars are not associated with a particular user, mail regarding the scheduling of the event cannot be sent. These calendars cannot receive mailed copies of schedules for the same reason.

Setting calendar preferences

To change the appearance of the Calendar display, select:

Options ⇨ **Preferences**

When you are through choosing Calendar preferences, press ⟨Enter⟩ or ⟨Ctrl⟩**X** to return to the **Calendar** menu.

```
──────────────────────── Preferences ────────────────────────
       Start time [8:00 a.m. ]
         End time [5:00 p.m. ]
         Interval [Even]   Half
      Time format [Standard]   24 Hour

      Skip Weekends and Holidays?      Yes   [No]
      Saturday and Sunday separate?    Yes   [No]
      Expanded daily display?          Yes   [No]
      Startup Calendar [perry          ]

      Save as default?  [Yes]   No
```

Here is a brief explanation of each of the fields in the Preferences form:

Field	Description
Start time	The time of day when you start scheduling events. The default is 8:00 a.m.
End time	The time of day when you stop scheduling events. The default is 5:00 p.m.
Interval	Display the calendar in one-hour increments ("Even", the default), or in half-hour increments ("Half").
Time format	Display the time in a.m. or p.m. format ("Standard", the default), or in 24-hour international format ("24 Hour").
Skip Weekends and Holidays	Skip ("Yes") or include ("No", the default) weekends and holidays when using the **Next** and **Back** options, and when using the **Search** command.
Saturday and Sunday separate	Place Saturdays and Sundays in separate boxes ("Yes"), or the same box ("No", the default), on the monthly calendar.
Expanded daily display	Include ("Yes") , or does not include ("No", the default), details about the event and who is invited to it in the daily calendar display.

(Continued)

Field	Description
Startup Calendar	The calendar invoked when you start the Calendar program. The default is the calendar corresponding to your user name. To switch calendars while running the Calendar program, select **Options ⇨ Switch.**
Save as default	Save the selections made in the Preferences form as the default ("Yes", the default), or return to the default preferences when you quit the calendar ("No").

Setting permissions on the current calendar

By default, the access permissions allow each user to have blind view, view, and schedule access to another user's calendar, and only allow the owner of a calendar full access. You can change the permissions to allow only certain users or groups of users (aliases) blind view, view, schedule, or full-access permission on your calendar by selecting:

Options ⇨ Configure

The Calendar displays the Configure form:

```
──────────────────────────── Configure ────────────────────────────────
 Automatic Notification
    Invitation and Cancellation Mail?   Yes    No
    Daily and Weekly Schedules?         Mail    Printout   [Off]

 Access Permissions
   Blind View [everyone                                              ]
        View [everyone                                              ]
    Schedule [everyone                                              ]
  Full Access [                                                     ]
```

Here is a brief explanation of each of the fields in the Configure form:

Field	Description
Invitation and Cancellation Mail	Set to "Yes" to specify that you want to be notified when others change your calendar. Also, if you cancel an event to which others were invited, a mail message, explaining the cancellation with up to two lines of descriptive text, is generated automatically. If set to "No", no mail is sent.
Daily and Weekly Schedules	Set to "Print" or "Mail" if you want to generate automatically a printed or mailed copy of your daily and weekly schedules. If you have chosen and saved the expanded daily calendar display in "Preferences" as default, then the copy will also be expanded.
Blind View	Specify who can see when events are scheduled but not the details.
View	Specify who can see the details of events.
Schedule	Specify who can add events.
Full Access	Specify who can change and delete events.

While you are editing the "Blind view," "View," "Schedule," and "Full Access" fields in the Configure form, you can display a point-and-pick listing of all existing calendar names. To do this, you must first use ⟨Ctrl⟩Y to delete "everyone" from the field you want to edit, then press ⟨F3⟩. You can then search the list for the name of a specific user by pressing ⟨F5⟩. To cancel the point-and-pick display, press ⟨Esc⟩. If you want to give permission to all users, enter "everyone" in the appropriate field. You may also enter the name of an alias in this field.

2

Deleting a calendar

Use **Erase** to clear all events from the calendar you are currently using. You might want to do this for a meeting room that is no longer available for use.

Once a calendar's events have been erased, they cannot be retrieved. You should thus be very careful when using **Erase**.

Before using this option, switch to the calendar you want to erase using **Option** ➪ **Switch** , then choose **Options** ➪ **Erase** .

Because you cannot undo this action once it has been confirmed, you are given two chances to change your mind and not erase the calendar.

If the calendar you deleted was your personal calendar, it remains on the screen with no events listed. Otherwise, it is removed entirely and you automatically return to your personal calendar.

Renaming a calendar

Rename allows you to rename any calendar made as described in "Creating a new calendar" (page 66) and for which you have full-access permission. You cannot rename your personal calendar.

To rename the current calendar, select **Options** ➪ **Rename** .

Erasing or renaming a calendar does not change the contents of the "Who" fields of any other users' calendars.

Managing aliases

An alias is a name used by the Calendar to refer to a group of users. Use **Alias** to create, modify, and delete calendar aliases. To use an alias, type the alias name in the "Who" field of the forms presented by the Calendar. When you schedule an event for an alias, mail is automatically sent to all the people on the alias, notifying them of the event.

There are two types of aliases: private and system. Private aliases are made and controlled by you, and only you have access to them. Only the system administrator can set up and edit system aliases.

To edit, delete, and create new private aliases, select:

Options ➪ **Alias**

When you select **Alias**, a list of your private aliases appears along with a menu. The items on the **Alias** menu allow you to:

- edit an alias

- add a new alias

- delete an alias

- view an alias without changing it

- switch to a private or a system alias list

- quit to the **Options** menu

Adding an alternative calendar to your Application List

The calendar that is first displayed when you enter the Calendar is specified by the name in the "Startup" field on the Preferences form; this is normally set to your own calendar. See "Setting calendar preferences" (page 67).

Additionally, you can use the procedure described in this section to specify a different initial calendar.

To specify another initial calendar, edit the entry for the Calendar program on the SCO Shell's Application list. (For detailed instructions, see "Adding an application to the list" (page 47) .)

Select **Options** ➪ **Applist** ➪ **Edit** and select **Calendar** from the list of applications displayed. Add one of the following options to the entry in the "Path Name" field:

Option	Description
-i *calendar_name*	Substitute *calendar_name* for the name of the calendar in the "Startup" field of the **Options** ➪ **Preferences** menu.
-r *calendar_name*	Substitute *calendar_name* for the name of the calendar in the "Startup" field of the **Options** ➪ **Preferences** menu, but display as "read only;" you cannot add, change or delete any information.

For example, to display user *alice*'s calendar when you first enter the Calendar program, change the "Path Name" entry to:

Path Name: [`$OALIB/calendar -i alice`   ]

To display the calendar for *sales* as read only when you first enter the Calendar program, change the "Path Name" entry to:

Path Name: [`$OALIB/calendar -r sales`   ]

You can also create an additional entry in the "Application List" to start a different calendar such as a departmental calendar or one that you are responsible for scheduling. To do this, select:

Options ⇨ **Applist** ⇨ **Add** ⇨ **Select**

To add the option needed to access the additional calendar, select:

Options ⇨ **Applist** ⇨ **Edit**

Resolving problems with Calendar information

If the Calendar program fails to get information from another computer, you may see a message indicating that there is no response from the calendar server. This message can indicate several problems: the computer containing the calendar data may be down, or the calendar server computer may be down or busy. Contact your SCO Shell administrator for assistance.

Using the Calculator

The SCO Shell Calculator mimics the operation and appearance of a simple electronic calculator.

The Calculator is available whenever you need to do some arithmetic from anywhere in SCO Shell and from any of the applications in SCO Shell.

The Calculator offers these features:

- addition, subtraction, multiplication, floating-point division, and percentage calculation

- up to four numbers can be held in memory

- simulated "paper tape" that records all the calculations you entered in your current session with the Calculator

- support for negative numbers

Note that you cannot use a mouse with the Calculator.

Starting the Calculator

To start the Calculator from the main SCO Shell menu, select:

Application ⇨ Calculator

You can use your terminal's numeric keypad as well as the number keys in the keyboard's top row. (If the keypad keys do not seem to work, try pressing ⟨Num Lock⟩.)

For example, type: **2 + 4 + 6 =**

There is no need to press ⟨Space⟩ between numerals. When you finish, the Calculator looks like this:

```
        Tape │                    Display │
             │                            │
             │                            │
             │                            │
  ┌──────────────────┬──────────────────────────────┐
  │           2  +   │                  12           │
  │           4  +   ├──────────────────────────────┤
  │           6  =   │                   ⎕Clr  ⎕Off  │
  │          12      │                              │
  │                  │   ▪7  ▪8  ▪9  ▪+  ▪-          │
  │                  │                              │
  │                  │   ▪4  ▪5  ▪6  ▪*  ▪/          │
  │                  │                              │
  │                  │   ▪1  ▪2  ▪3  ▪%  ▪Neg        │
  │                  │                              │
  │                  │   ▪0  ▪.  ▪=  ▪Sto ▪Rcl       │
  └──────────────────┴──────────────────────────────┘
```
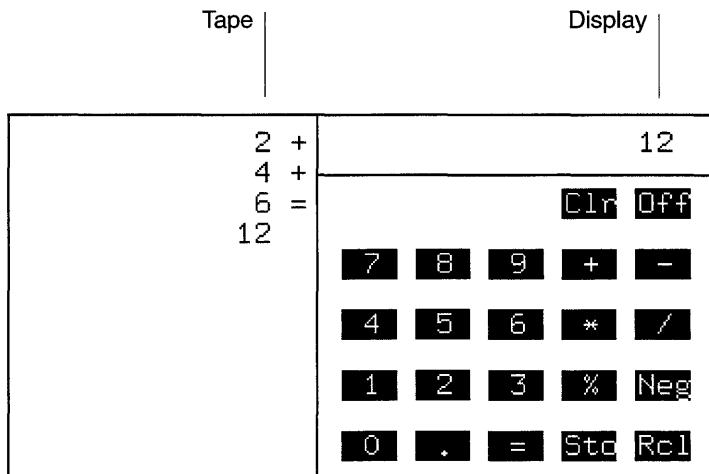
The display acts just like the one on a pocket calculator. You enter asterisk "*" to multiply numbers together (this is equivalent to the "×" key on a pocket calculator). The Calculator keeps a running total of the calculation until you reach the end. (If you want, you can press ⟨Enter⟩ instead of the "=" character; they both do the same thing.)

The tape keeps a record of the whole problem, just as you entered it. The answer does not appear there until you press =, just as on an adding machine tape. The arrow keys let you rewind the tape to take a look at the calculations that you did earlier.

Calculator commands

You enter calculations on the Calculator by typing them in. You do not choose commands or numbers by highlighting parts of the screen, as you might in other SCO Shell applications.

The keys that you see on the screen are reminders or prompts for the Calculator commands. The actual commands are the first letters of the function key names. For example, to press the Off key, you type **o**; to press the Sto (store) key, you type **s**; and so on. The following section details the Calculator's features, including the use of the scrolling tape, the memories, and the percentage key.

Using the Calculator's features

The following sections explain the Calculator's features in more detail.

Simple arithmetic

Here are some simple problems you can enter on the Calculator.

    5 * 4 =
    100 + 4 + 6 + 7 - 3 / 2 =
    1000 / 5 / 10 / 30 =
    1.25 + 3.85 / 1.7 =

Type the numbers and arithmetic symbols on your keyboard. Remember to press = or ⟨Enter⟩ to finish each problem. When you begin a new problem, the total from the last one clears from the display automatically.

Scrolling the tape

Enter several long calculations, until the earliest ones have moved up the tape and off the top of the calculator. Now press the ⟨Up Arrow⟩ key repeatedly to "rewind" the tape. The earlier calculations reappear.

Use the ⟨Down Arrow⟩ key for the opposite effect, to advance the tape back to more recent calculations.

To move through the tape more quickly, use ⟨Ctrl⟩**U** and ⟨Ctrl⟩**D** to rewind and advance the tape five lines at a time; ⟨PgUp⟩ and ⟨PgDn⟩ to rewind and advance the tape ten lines at a time.

If you want to rewind the tape all the way, press ⟨Home⟩ or ⟨Ctrl⟩**T**. Pressing ⟨End⟩ does the opposite: it advances the tape to the last calculation entered.

With the tape still "rewound," enter a new calculation and see what happens: the tape automatically advances itself to the end of the most recent calculation, just as if you had pressed ⟨End⟩.

The figures on this tape are kept only while the Calculator is on. When you turn off the calculator, the figures disappear permanently.

The Calculator only remembers the previous 100 calculations on the tape. You cannot view any calculations previous to the first on the tape.

Storing numbers in memory

You can store up to four numbers in the Calculator's memory for later use. For example, you may want to use the results of a calculation in several others. Rather than re-entering the figure for each calculation, store the figure in memory and recall it as you need it.

Storing a number

The number you want to store must be on the Calculator's display, as either the answer to the previous calculation or simply a number you typed in. Follow this procedure:

1. Enter the following calculation: **5+6*3-3*4=**

   The result is 120.

2. Press **s** (Sto).

   The Store box holds four different fields: A, B, C, and D. The first field, A, is already highlighted. Each field can hold one number. Press ⟨Enter⟩ to store 120 in field A.

3. After a short delay, the Calculator screen returns to normal.

To store another number in memory, repeat the procedure but use the arrow keys to highlight a different field in the Store box. If all four fields already hold figures, just highlight one that holds a figure you no longer need.

Using a stored number

Now that you have stored a number, suppose that you want to use it in a problem. For example, **104 / 4** * *stored_number*. Follow this procedure:

1. Enter **104/4*** in the calculator. After you enter the "*" symbol, press **r** (Rcl).

   The Recall box shows the number currently stored in memory. Press ⟨Enter⟩ to recall it. (If there are several numbers in the recall box, use the arrow keys to highlight the one you want to recall.)

2. The Recall box disappears at once. The stored number is now in the display window, just as if you had entered it by hand. Enter = to complete the calculation. Your answer is 3120.

   The number you retrieved is still in memory, and it stays there until you replace it with another one or turn off the Calculator.

Negative numbers

To enter a negative number, type the number followed by an **n**. The number is now preceded by a minus sign on both the display and the tape. It behaves as a negative number in all calculations. Pressing **n** also removes a minus sign if the displayed value already has one.

For example, type **5** * **5n =** to multiply 5 by negative 5. The result is -25.

Percentages

You may sometimes perform calculations whose operands include both a number and a percentage of that same number. The Calculator's percentage key lets you do this without calculating the percentage separately.

Suppose that you make a $25.00 purchase and want to calculate the total price when six percent sales tax is included. You enter **25 + 6 %** and the result is 26.5, or $26.50. Note that you do not need to press **=**.

If you enter the wrong number

If you enter the wrong number in the middle of a problem, type **c** to clear the number and enter a new one. For example, if you want to multiply 7 by 7, but enter 7 times 6, type the following, **7** * **6 c 7** and the mistake is corrected.

The ⟨Bksp⟩ key also helps you to correct mistakes. If you want to change or erase the number you just typed, press ⟨Bksp⟩ repeatedly to make the number disappear one digit at a time.

Exiting the Calculator

To turn off the Calculator, press **o**, or ⟨Esc⟩. You then return to the program you called the Calculator from.

All the numbers stored in memory or on the tape now disappear, just as on a real pocket calculator.

As with all open programs, you must call up and quit the Calculator when you finish work for the day before logging off your computer.

Working at the
Shell Prompt

Chapter 3

Working with files and directories

This chapter discusses files and directories and the basics of working at the command prompt. It describes how to:

3

Getting to the command prompt

The commands described in this book are all intended to be typed at a command prompt in a text display (or window, if you are using a graphical terminal).

There are several ways of getting to a command prompt, depending on how your system is configured. In general, you must first log in to the system, then (if you are using a graphical environment) open a shell window.

If you are using the graphical environment:

1. Log in, as described in "Starting the Desktop" in the *SCO OpenServer Handbook*.

2. Open a UNIX window by double-clicking on the UNIX icon, located by default on the Desktop. Otherwise, select **UNIX** from the **Tools** menu.

If you are working on a character terminal:

1. Log in, as described in "Logging in" in the *Operating System Tutorial*.

2. If your login automatically starts up the SCO Shell, type:

    !ksh⟨Enter⟩

You should now see a "$" or "%" symbol with the cursor positioned next to it. This is the command prompt: you are now ready to start entering commands.

Note that the examples in this chapter assume that you are running the Korn shell. If you are not sure which shell you are running, see "Identifying your login shell" (page 224). If you are not running the Korn shell, type:

    ksh⟨Enter⟩

This starts a temporary Korn shell session.

If you have not used the command prompt before, you may want to refer to Chapter 1, "Getting started" in the *Operating System Tutorial* before continuing with this chapter.

Files and directories

The basic unit in which the SCO OpenServer system stores information is the *file*. A file is a named collection of data that you can move around, copy, rename, or delete. Files are stored in a *filesystem*, a storage area on your computer's hard disk or disks. A filesystem is split into *directories*, which are smaller storage areas that make it easier to locate individual files.

Using files

The system is unconcerned with the structure of the information in a file; all it sees is a stream of characters. Individual programs may impose a structure on the file, and you may see references to records and fields within a file, but the file itself is the smallest piece of information that is stored under a name and recognized by the system.

Using directories

A typical system contains many files, perhaps tens of thousands. To keep track of them, they are divided into directories. A directory is an area of the filesystem that is assigned a name; it can contain files and, optionally, directories. By using the name of a directory instead of the name of a file as the parameter of a command, you can make the command operate on all the files stored in that directory simultaneously. The first, top level, directory on the system is called the *root* directory; all the other directories and files in the system trace their ancestry back to it.

Files belonging to a particular user are usually stored in that user's own directory; those associated with a single project or application are also often stored in a single directory. Users can also create directories within their home directory to store files relating to specific projects. The operating system looks after the organization of system files, but you are responsible for the organization of your own files.

File and directory attributes

The system handles files and directories in the same way; directories are just specialized files, containing other files and directories rather than program code or text. Files and directories both have a name, a path, and a set of attributes. Internally, the system keeps track of files and directories using *inodes*, or index nodes. See "How the system manages files and directories" (page 83).

3

A simple way of checking some of the attributes of a file or directory is the *long listing*, obtained using the -l option to the **ls**(C) command (or just **l**):

```
                                        Date of last
File type    Number of links    Group   modification      Filename
   |              |               |          |                 |
                          Size           Time of last
       Permissions |   Owner    in bytes   modification   |
   |         |       |      |      |         |    |         |
   -rwxrwxrwx 1 perry techpubs 648509 Jul 26 08:15 minutes
   -rw-r-wr-x 1 perry unixdoc    2256 Jul 25 10:23 agenda
   drwxr-xr-x 2 perry techpubs     48 Mar 02 18.51 bin
```

What you see in a long listing

- The first field (file type) indicates the sort of file that is present in the listing. The following codes are some of those used (for a full list, see **ls**(C)):

  - ordinary file

  b block special file

  c character special file

  d directory

  l symbolic link

  p named pipe

- The second field (permissions) shows who is permitted to read, write, or execute a file, or change to a directory. Users are split into the file's owner, people in the same work group as the owner, and other people. A separate set of permissions is maintained for each category. The notation used here is explained in detail in "Access control for files and directories" (page 121).

- The third field (links) shows the number of links that exist for the file (links are discussed in "Creating links to files and directories" (page 94)).

- The fourth field (owner) shows the login name of the owner of the file.

- The fifth field (group) shows the group to which the file belongs; that is, the group of users who have "group" access permission to the file. See "Finding out your group" (page 126) for an explanation of groups.

- The sixth field (size) shows the number of bytes in the file.

- The seventh and eighth fields (date and time of last modification) show the date and time when the file was last modified.

- The final field (filename) shows the name of the file. See "Filenaming conventions" (page 83) for more on filenames.

How the system manages files and directories

Internally, the system keeps track of files and directories using *inodes*. An inode (or index node) is a representation of a file that stores all the data belonging to that file, such as owner, type, size, access permissions, access times and the file's layout on disk. Each inode has a unique number which is used by the system in file handling operations: the filename is simply a device to make the filesystem easier to use for humans. In fact, while a file may have only one inode number, it may have several filenames, these being *links* to the one inode. See "Creating links to files and directories" (page 94) for more information on links to files and directories.

Filenaming conventions

The maximum permitted length of a file or directory name is 255 characters. In fact, this is controlled by the value of the {NAME_MAX} constant; to check the value of this, use the **getconf**(C) command, as follows:

```
$ getconf NAME_MAX .
255
```

Pathnames, which are described in "How directories are organized" (page 84), have a maximum permitted length of 1024 characters (as controlled by {PATH_MAX}, which is also controllable using **getconf**).

An important consideration, where Open Systems are an issue, is filename portability. Many of the international standards specify a character set that should be used for the construction of portable filenames. The IEEE POSIX standard, for example, specifies the following Portable Filename Character Set:

- the uppercase letters (A-Z)
- the lowercase letters (a-z)
- the decimal digits (0-9)
- the dot (.), the hyphen (-) and the underscore (_)
- the slash (/): this is a special case which is discussed below

Portable filenames should not begin with a hyphen, although it may appear in any other position. Specifically, the following characters should not be used in file or directory names because they have a special meaning for the UNIX system:

```
! " ' ' ; / $ < > ( ) | { } [ ] ~
```

The slash character (/) signifies both the *root* directory and the pathname element separator, and is valid only in these contexts. See "How directories are organized" (page 84) for more information on pathnames.

Filenames may begin with a dot (.), but this has the effect of excluding them from normal directory listings. See "Listing the contents of a directory" (page 87) for details of how to list these "hidden" files.

Managing directories

Together with files, directories constitute the UNIX filesystem, which has a characteristic inverted tree structure beginning at the *root* directory. As with files (see "Managing files" (page 101)), you can create, rename, copy, erase, and compare directories. You can also set access permissions on directories to prevent or enable their use by other users. This is explained in "Access control for files and directories" (page 121).

How directories are organized

Directories are organized as an inverted tree structure. Only one directory, at the top of the tree, is not contained in any other directory. This is called the *root* directory, and its name is represented by a slash (/) character. To help clarify this, look at the following picture of part of a directory tree:

```
                        / (root)
                            |
                           usr
                            |
       ┌──────────┬─────────┴─────────┬──────────┐
     johnd      sarah             susanna       perry
                            ┌────────┬──────────┬─────────┐
                        dead.letter review     mail      mbox
```
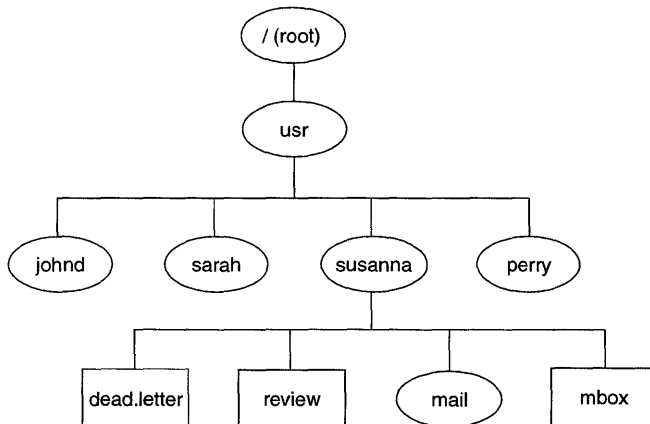
In this picture, directories are depicted by ovals, and ordinary files by rectangles. Notice that most of the directories have lines coming out of them, indicating that they lead to files or other directories.

When you work within a UNIX filesystem, you always have a *current working directory*. All filenames and commands that you type at the prompt are evaluated with respect to this position. When you log in, your current working directory is set to the directory created for your user account. This location is known as your *home* directory.

Any file or directory on the system can be specified uniquely by its *pathname*. Pathnames are instructions for finding a file or directory; they list, in order, each directory you must pass through to get to the file or directory in question. When the pathname is written down, the directories are separated by slashes (/). For example, the full pathname of the file called *review* in the diagram above is */usr/susanna/review*. The first slash character denotes the *root* directory: all the others are separators.

A pathname that begins at the *root* directory is called an *absolute* or full pathname. There are also *relative* pathnames, which give directions to a file relative to your current working directory. Two dots in a row (..) represent the *parent* of the current directory.

For example, if you were working in the directory called */usr/sarah* and you needed to specify the file in */usr/susanna* called *mbox*, you could use the relative pathname *../susanna/mbox*, or the absolute pathname */usr/susanna/mbox*. (Remember, ".." refers to the parent directory of the current working directory; so in this case it refers to */usr*, which is the parent of both *susanna* and *sarah*.)

An example: what the system contains

When an SCO OpenServer system is installed, many directories are created automatically. The following figure shows a partial structure of a UNIX root filesystem. (A full root filesystem would be too large to show here.)

/ (the root directory)

| | | | | |
| bin | | dev
(device files) | tmp
(system
scratch pad) | | tcb
(trusted
computer
base or user
security files) | unix
(the kernel) |

etc
(system files
and commands)

usr

default
(system
default
files)

auth
(system
security)

bin
(more basic
commands
plus any
application
commands)

lib
(application
library files)

spool
(application
queued files)

3

/	The *root* directory is the root of the filesystem tree. Every directory is a subdirectory of *root*.
/bin and */usr/bin*	These directories contain most of the UNIX system commands. Generally, standard UNIX commands and applications are held in */bin*, whereas group-specific commands and applications, that is, those used by a particular group of users, are held in */usr/bin*.
/dev	This directory contains all the special device files. Special device files are access points to all the peripherals connected to the system.
/etc	This directory contains many of the system configuration files and system administration commands.
/unix	This file contains the UNIX kernel program. This program is loaded into memory when the operating system starts up. It is the heart of the SCO OpenServer system; for more information see "The UNIX system kernel" (page 397).
/usr/lib	This directory contains many application library files.
/usr/spool	This directory is used by many commands for storing temporary files or files in a queue.
/var/opt/	This directory contains *storage sections*. See "Creating a link to a directory" (page 96) for information about symbolic links and storage sections.

Creating a directory

To create a new directory, use the **mkdir**(C) (make directory) command, as follows:

>   **mkdir** *directory*

The *directory* argument can be either a simple name, in which case the new directory is created within the current working directory, or a pathname. For example, if you want to create a new subdirectory called *projects* in the directory called */u/workfiles*, you can do the following:

```
$ mkdir /u/workfiles/projects
```

You get the same result by entering the following:

```
$ cd /u/workfiles
$ mkdir projects
```

The **cd** command stands for "change directory". See "Navigating the filesystem" (page 92) for more details.

For details of naming conventions for directories, see "Filenaming conventions" (page 83).

You can create a directory within any directory where you have write permissions. See "Access control for files and directories" (page 121) for details of how to manage access to files and directories.

You can create several directories at the same level. For example, to create directories called *directory1* through *directory3*, use the following command:

```
$ mkdir directory1 directory2 directory3
```

If you need to create an entire directory path at once, use the **mkdir -p** (path) option. The following command creates a directory called *user_guide*, and any of the other directories in the specified path that do not already exist:

```
$ mkdir -p projects/myprojects/user_guide
```

Bear in mind that the efficiency of the filesystem has some impact on overall system performance. For example, you should not let your directories grow larger than necessary. Ideally, a directory should contain no more than 640 files (providing that the number of characters in each filename is 14 or less), otherwise it may take the system longer to search the directory whenever you access a file stored in it. If you use longer filenames, the limit may be lower. File size is also significant: large files impose overheads on access. See "Looking at the contents of a file" (page 102) for more information.

Listing the contents of a directory

The names and other information about the files and subdirectories contained within a directory can be displayed using the **ls**(C) family of commands. In its simplest form, **ls** gives a list of the filenames found in the current working directory, as follows:

```
$ ls
cs-save
gav_make
glossary.s
graphics
intro.err
nohup.out
procs.txt
```

To see a list of the filenames in a multi-column format, use the **lc** variant, as follows:

```
$ lc
cs-save          intro.err
gav_make         nohup.out
glossary.s       procs.txt
graphics
```

3

For a full listing, giving file size, permissions, owner and other items of information, use the **ls -l** option, as follows:

```
$ ls -l
drwx------   2 chris   techpubs       64 Jul 07 17:19 tools
drwxr-xr-x   2 chris   techpubs       80 Jul 06 16:51 trash
-rw-r--r--   1 chris   techpubs     6204 Sep 23 09:34 travel
```

For a complete breakdown of this information, see "File and directory attributes" (page 81). In fact, this version of the command is used so commonly, that it can be entered in shorthand, as l(C).

As we saw in "Filenaming conventions" (page 83), filenames may begin with a dot, in which case, the files are hidden from normal directory listings. The **ls -a** (all) option displays hidden files as well as normal files, as follows:

```
$ ls -a
.
..
.history
.kshrc
.mailbox
.profile
cs-save
gav_make
glossary.s
graphics
intro.err
nohup.out
procs.txt
```

To list the contents of another directory, without first moving to that directory, use the **ls** command, specifying the directory to look at as an argument, as follows:

```
$ ls /u/workfiles/projects
```

This command line lists the contents of a directory called */u/workfiles/projects*.

You need permission to read a directory before you can view its contents. See "Access control for files and directories" (page 121) for an explanation of permissions.

The tilde-plus sequence (˜+) is expanded by the shell to point to the current working directory (actually, the value of the **PWD** environment variable). A more useful variant of this notation is the tilde-minus notation (˜-), which expands to the value of **OLDPWD**, that is, the previous working directory. This allows you to refer back to your earlier work without having to type in the relevant pathname, as follows:

```
$ ls -l
drwx------    2 chris    techpubs      64 Jul 07 17:19 tools
drwxr-xr-x    2 chris    techpubs      80 Jul 06 16:51 trash
-rw-r--r--    1 chris    techpubs    6204 Sep 23 09:34 travel
$ cd ../project2
$ ls -l
-rw-r--r--    1 chris    techpubs    3137 Oct 24 17:49 agenda
drwxr-xr-x    2 chris    techpubs      96 Aug 31 13:08 bin
$ ls -l ˜-
drwx------    2 chris    techpubs      64 Jul 07 17:19 tools
drwxr-xr-x    2 chris    techpubs      80 Jul 06 16:51 trash
-rw-r--r--    1 chris    techpubs    6204 Sep 23 09:34 travel
```

If you list the contents of a directory that contains more files and subdirectories than can be displayed on one screen, the list scrolls continually until all the files have been displayed. This makes it very difficult to read them. To view the list one screen at a time, type the following:

```
$ ls | more
```

The output from **ls** is *piped* to the **more**(C) command which then displays it; **more** prints its input one screen at a time. (See "Running commands in a pipeline" (page 120) for more information about pipes.) Press ⟨Enter⟩ to scroll down by one line, or the ⟨Space⟩ bar to scroll down by one screen. Otherwise, you can pipe the output from **ls** into the **pg**(C) command, which performs a similar operation. The main difference between the two is that **pg** allows you to step backward through a file (by pressing the minus key (-)), as well as forward (by pressing ⟨Enter⟩ or the plus key (+)).

Another way to pause the scrolling is to use the ⟨Ctrl⟩**S** and ⟨Ctrl⟩**Q** keystrokes. Press ⟨Ctrl⟩**S** to temporarily stop the scrolling, and ⟨Ctrl⟩**Q** to continue. If you want to stop the listing completely, press ⟨Del⟩. These keystrokes depend on your terminal setup; if they do not seem to work, ask your system administrator to help you.

3

Renaming a directory

To rename or move a directory, use the **mv**(C) (move) command, as follows:

mv *oldname newname*

oldname is the directory's current name, and *newname* is the new name you want to assign it. As with the **mkdir** command, arguments to **mv** may be simple names or pathnames, as follows:

```
$ mv users_guide uguide
$ mv appendix ../docset2/admin_guide/app
```

The first of these commands renames a subdirectory: the second renames a directory from *appendix* to *app*, at the same time moving it to another location.

Copying a directory

To copy all the files in a directory, use the **copy**(C) command, as follows:

copy *old_directory new_directory*

old_directory is the name of the directory you want to copy, and *new_directory* is the name you want the copy to have. Simple names and pathnames are acceptable in both cases.

In order to copy a directory and all its files, you must have read and execute permissions on that directory, read permission on the files in that directory, and write permission on the directory into which you want to copy. See "Access control for files and directories" (page 121) for details.

Removing a directory

To remove an empty directory, use the **rmdir**(C) command, as follows:

rmdir *directory*

This will fail if there are any files or subdirectories in the directory. If this is so, you must either delete the files it contains, or move them to other directories. The only exceptions are the dot and dot-dot directories, which you cannot delete, and which are dealt with by the UNIX system itself.

You can remove a directory and any files it contains by using the **rm -r** option, as follows:

rm -r *directory*

Be very careful when doing this because the **-r** option tells **rm** to *recursively* enter any subdirectories and remove their contents. You may remove more than you expect. It is often safer to use the **rm -i** option. See "Removing a file" (page 105) for information on interactive deletion. You must have write permission on a directory before you can remove it.

Comparing directories

Comparing directories is useful when two people are working on the same set of files, for example, the chapters of a book. By comparing the directories you can quickly identify which files are different.

To compare two directories, use the **dircmp**(C) (directory compare) command, as follows:

dircmp *directory1 directory2*

The output is a list of the differences between the directory listings for *directory1* and *directory2*, for example:

```
$ dircmp ./dir1 ./dir2
Jan 12 10:54 1995  dir1 only and dir2 only Page 1

                                       ./t.appx.s
                                       ./t.prog.s
                                       ./t.scosh.s
                                       ./t.shell.s
     .
     .
     .
Jan 12 10:54 1995  Comparison of dir1 dir2 Page 1

directory        .
same             ./0.ct.s
same             ./00.partno.s
same             ./00.title.s
     .
     .
     .
```

The top of the listing consists of those files which are unique to one or other of the directories; in this example, *dir2* contains four files which are not present in *dir1*. The listing then contains a detailed comparison of every file which is named in both directories.

3

Navigating the filesystem

The following sections explain how to find out where you are in the directory tree, how to move from directory to directory, and how to list the contents of a directory.

Finding out where you are in the system

After a number of **cd** operations, it is possible to lose track of where you are in the filesystem. To identify your current directory, use the **pwd**(C) (print working directory) command. This command takes no arguments.

The output from **pwd** shows the absolute pathname of your current directory. For example, if your login is *johnd* and you are in your home directory (which is a subdirectory of */usr*), the output would probably look like the following:

```
$ pwd
/usr/johnd
```

If you are using the Korn shell (see "Identifying your login shell" (page 224) if you are unsure about this), you will find it useful to issue the following command as soon as you log in, or add it to your login script:

```
alias pwd='pwd -P'
```

The reason for this is explained in "Navigating symbolic links" (page 97).

Changing directory

Once a directory system exists, you need to know how to get from one directory to another, thereby changing your current working directory. This is done using the **cd**(C) (change directory) command. As an argument to the command, you specify the directory you want to change to, as follows:

cd *directory*

For example, if your current directory is */u/johnd*, you can to change to a directory called */u/workfiles/projects* by specifying its absolute pathname, as follows:

```
$ cd /u/workfiles/projects
```

You can also change to the directory by specifying its relative pathname, as follows:

```
$ cd ../workfiles/projects
```

Note that you must have execute permission on a directory before you can change to it. See "Access control for files and directories" (page 121). See "Returning to your home directory" (page 93) for details of a special usage of **cd**.

Returning to your home directory

You can return to your home directory from anywhere in the directory structure by typing **cd** on its own.

Your home directory is stored in the **HOME** environment variable. You can display this value using the following command line:

```
$ echo $HOME
```

Remember to include the dollar sign in the command line: without it, the **echo**(C) command will simply return the word "HOME". For an explanation of environment variables, see "Understanding variables" (page 226). For more on the **echo** command, see "Forcing a program to read standard input and output" (page 119).

A useful tool for filesystem navigation is the tilde character (˜). The shell expands a tilde to the absolute pathname of your home directory. This notation can be included in a pathname, as follows:

```
$ pwd
/tmp
$ cd ˜/i486/dev/backup/scripts
$ pwd
/usr/martins/i486/dev/backup/scripts
```

Because the pathname is absolute, you do not need to know where it is in relation to your current working directory.

The tilde can also be used in conjunction with other users' login names, acting as a shorthand way of accessing their files without necessarily knowing exactly where their home directories are located. For example, the following command line allows you to check a colleague's directories for any file called *hyacinth*:

```
$ find ˜john -follow -name hyacinth -print
```

(See "Finding files" (page 113) for how to use the **find** command.)

3

Creating links to files and directories

You may need to make a file accessible from more than one directory, and by more than one user, but still keep it as a single file. This is often the case when you need to share the data in a file with your colleagues. To prevent different versions of the file from proliferating, only one copy of the file exists, but *links* are created that allow you and your colleagues to access the file from your home directories or another convenient location. (Note that there is the danger of a file becoming corrupted if more than one person tries to edit it at the same time.)

Creating a link to a file

To create a link to a file, use the **ln**(C) command, as follows:

ln *filename linkname*

For example, suppose you have a file called *user_guide* which is located in */u/workgrp/tasks/projects*. To work on this file you would normally **cd** to that directory before opening the file. However, by creating a link to the file, you can access it from your current directory (without needing to enter the full path of the file). To do this, type the following:

    $ ln /u/workgrp/tasks/projects/user_guide my_guide

The *my_guide* argument identifies the link. Whenever you want to work on the file, which is now known to the system by two names (*user_guide* and *my_guide*), you can access it from the current directory by using *my_guide* as the filename.

You must have write permission on a directory before you can create a link that involves that directory or a file in that directory. You cannot create a hard link (the kind of link described above) to a directory or a file on a different file-system. To create a link to a directory or a different filesystem, you must use a *symbolic* link. See "Creating a link to a directory" (page 96) for details.

Links can be removed using **rm**. If a file has several links, it is not physically deleted until the the final link is removed.

Finding out whether a file has hard links

To find out how many hard links there are to a file, use the **ls -l** or **l** (long listing) command, as follows:

```
$ l
-rw-r--r--   1 johnd    unixdoc    10586 Feb 25 12:26 1.start
-rw-rw-r--   2 johnd    techpubs   61339 Feb 24 14:45 2.scosh
-rw-rw-r--   1 johnd    techpubs   14741 Feb 25 11:18 3.dire
-rw-rw-r--   3 johnd    techpubs   40419 Feb 25 15:57 4.files
```

You need to locate all the links to a file or directory if you want to delete it: as long as there are links, you cannot delete the file or directory.

In a long listing, the number of links are shown in the second column from the left (after the sets of permissions). For example, in the above example, the file *4.files* has three links to it.

You can find out where the common links to a single file are located, in two steps. First, you need to identify the *inode number* of the file (see "How the system manages files and directories" (page 83) for information on inodes). To do this, use the **ls -i** option, as follows:

```
$ ls -i
20350 basking_shark
 3886 cod
 2002 halibut
 3526 herring
10182 narwhal
```

The number before the filename is the file's inode number, for example, 2002 for the file called *halibut*.

To trace all the links to this file, you must find all the other files in the filesystem with this inode number. You can do this using the **find -inum** option, as follows:

```
$ find / -inum 2002 -print 2>/dev/null
/u/dave/tmp/halibut
/u/charles/fish6
/u/michael/project2/ichthyo14
```

In this case, there are three files with the same inode number. In order to delete this file from the filesystem, it would be necessary to run **rm** on all the links.

Creating a link to a directory

It is often useful to change to another directory without typing its full path-name: *symbolic* links provide a useful shortcut to do this. A symbolic link differs from a hard link. It is a small file that contains a reference (by name) to a directory or file that already exists. Unlike normal links, symbolic links can cross filesystems and link to directories. (They are used extensively by the system.) Also unlike normal links, symbolic links are separate files; they cease to work if the file they point to is deleted or renamed, or if they are moved.

Many of the files found in */bin*, */lib*, and */usr* are actually *symbolic links* that point to files (of the same name) stored below */var/opt*. The directories these files are located in are called "storage sections". Storage sections are used because they make it easier to install system upgrades. Software subsystems (such as UUCP) consist of many files, which may be installed in several directories. However, all the files in a subsystem belong to a single storage section. By overwriting the contents of the (single) storage section directory, all the files in the subsystem can be updated simultaneously.

Symbolic links are identified in a directory listing by a " -> ", as follows:

```
$ 1
drw-r--r--  1 johnd  unixdoc    29  Feb 27 15:56 mydata -> /u/work
group/tasks/project/01
```

You can obtain a directory listing without symbolic links visible in it by specifying the **-L** (logical) option. This makes **ls** (or **l**, or any related program) list the directory, replacing the information about each symbolic link with the details for the file pointed to by the link:

```
$ 1 -L
drw-r--r--  1 johnd  unixdoc  10297  Feb 27 15:56 mydata
```

To create a symbolic link, use the **ln -s** option, as follows:

```
ln -s directory symbolic_link
```

For example, suppose you work in */u/workgrp/tasks/projects* and your home directory is */u/me*. Your normal command to work on a file would be the following:

```
$ cd /u/workgrp/tasks/projects
```

To reduce the typing required, enter the following command:

```
$ ln -s /u/workgrp/tasks/projects mydata
```

This command creates a symbolic link called *mydata* in your current directory. From now on, *mydata* and */u/workgrp/tasks/projects* refer to the same location, and you can relocate to */u/workgrp/tasks/projects* by typing **cd mydata** instead of typing in the full pathname.

You must have write permission on a directory before you can create a link that involves that directory or a file in that directory.

See also "Access control for files and directories" (page 121).

If you remove a symbolic link, only the link itself is removed. If you remove (or move) the directory or file to which the link points, the link will be left pointing to nothing.

Navigating symbolic links

Because the system uses symbolic links extensively, you may encounter problems in identifying your current working directory. For example, suppose you create a symbolic link to a directory, then change directory using the new link, as follows:

```
$ ln -s /u/workgrp/tasks/projects mydata
$ pwd
/u/people/mike
$ cd mydata
$ pwd
/u/people/mike/mydata
$
```

In this example, you create a symbolic link called *mydata*, pointing to */u/workgrp/tasks/projects*. However, if you change directory via the link *mydata*, you actually see yourself as being in */u/people/mike/mydata*. This is the *logical* present working directory; that is, the path the user traversed to reach the directory. The directory also has a *physical* path, that is, the actual pathname of the current directory, relative to the top of the filesystem.

Now you are in */u/workgrp/tasks/projects*. Suppose you create another symbolic link and change directory into it:

```
$ ln -s /u/people people
$ pwd
/u/people/mike/mydata
$ cd people
$ pwd
/u/people/mike/mydata/people
$
```

When you change directory into *people*, you are following a link to */u/people*. This is higher in the directory tree than your original starting point, but because you are traversing another symbolic link, your logical present working directory is another level down the tree.

If you then type **cd ..** (to go up a directory), where you end up depends on your shell. If you are running the Korn shell, the **cd ..** command goes up a level in your logical directory path: */u/people/mike/mydata* becomes your present working directory. If you are running any other shell, the **cd ..** command goes up a level in your physical directory path: */u* becomes the present working directory.

Despite the apparent complexity, it is possible to determine your physical working directory in a directory tree populated with symbolic links. There are two techniques, for Korn shell users and for others:

Korn shell users	Use the command **pwd -P** to identify your absolute current working directory. (The command **pwd** is built into the Korn shell. It normally returns the logical present working directory; the **-P** option makes **pwd** return the physical path to the current directory.
others	Use the command **pwd**(C). The non-Korn shells do not include a built-in **pwd** command. The external **pwd** command returns the physical path to the current directory by default.

In general, directory traversal commands built into the Korn shell accept two options, **-P** and **-L. -P** makes the command refer to the physical working directory, while **-L** makes the command refer to the logical working directory (that is, to the path taken through any symbolic links).

The equivalent commands, in any other shell, always apply to the physical working directory.

Mounting a filesystem

In addition to containing loose files, a storage medium may be used to store a whole filesystem, with its own root directory and subdirectories. A filesystem stored in this way can be *mounted* onto the main filesystem using the **mnt**(C) command.

Once a filesystem has been mounted on another filesystem, its root directory appears as a subdirectory of the parent filesystem. You can enter the mounted filesystem by using **cd**, and it looks just like another subdirectory hierarchy, even though it resides on another physical device.
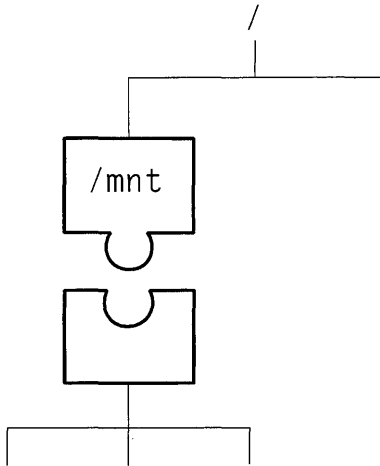
Mounting a filesystem

To mount a CD-ROM filesystem, for example, you must first place the disk in a CD-ROM drive installed on your computer, then use the **mnt** command. For example:

```
$ mnt /mnt
```

The */mnt* directory is a mount point; when a device is mounted on it that device's root directory appears in the filesystem instead of */mnt*. Any files that existed in */mnt* before the new filesystem was mounted on it are obscured, although they will be accessible again when the filesystem is unmounted.

Once the CD-ROM is mounted, it appears on the system as a tree of subdirectories, with the root directory of the CD-ROM located in the mount directory */mnt*.

The **mnt** command reads a file called */etc/default/filesys* (see *filesys*(F)), which contains a list of mountable filesystems. This file also specifies the name of the device associated with the filesystem (the `bdev` keyword) and the absolute pathname of the filesystem's mount point within the parent filesystem (the `mountdir` keyword).

Your system administrator must have added an entry to this file before you can use **mnt** to mount a CD-ROM. You can examine the contents of */etc/default/filesys* with the **mnt -t** option, as follows:

```
$ mnt -t
Mount Directory:                /apps
Block Device:                   /dev/apps
Character Device:               /dev/rapps
Password required:              No
Mount if requested:             No
Fsck if requested:              Only if filesystem is dirty
    Fsck flags:                 -y
Mount at system startup:        Yes
Fsck at system startup:         Only if filesystem is dirty
    Fsck options:               -y
Run command before mounting:    No
Run command after mounting:  ·  No

Mount Directory:                /private
Block Device:                   /dev/private
Character Device:               /dev/rprivate
Password required:              No
Mount if requested:             Yes
Fsck if requested:              Only if filesystem is dirty
    Fsck flags:                 -y
Mount at system startup:        Yes
Fsck at system startup:         Only if filesystem is dirty
    Fsck options:               -y
Run command before mounting:    No
Run command after mounting:     No
```

Otherwise, use **cat**(C) to show the contents of */etc/default/filesys*, as follows:

```
$ cat /etc/default/filesys
    .
    .
    .
bdev=/dev/apps cdev=/dev/rapps \
        mountdir=/apps rcmount=yes \
        mount=no fsckflags=-y
bdev=/dev/private cdev=/dev/rprivate \

        mountdir=/private rcmount=yes \
        mount=yes fsckflags=-y
```

For a filesystem to be mountable by a user other than the root user, its */etc/default/filesys* entry must contain the command mount=yes. In the example, the filesystem */dev/private* is mountable by users while */dev/apps* is not.

To unmount a filesystem, use either the **umnt**(C) command, as follows, or the **mnt -u** option.

```
$ umnt /mnt
```

You can also mount a DOS filesystem, allowing the use of DOS files without first copying them into the UNIX filesystem. For details, see "Using mounted DOS filesystems" (page 179).

Managing files

The directory handling operations (creating, deleting, administering and navigating) allow you to impose a structure on your area of the filesystem. It is the files, however, that store information like program code, text, database records and graphics. The SCO OpenServer system supplies a wide range of commands for managing files. The following sections discuss some of these. Some of the more complex systems are discussed in other chapter; see, for example, "A quick tour of vi" (page 132) for an explanation of how to use the **vi**(C) editor.

Finding out what type of data a file contains

As we saw in "File and directory attributes" (page 81), the system supports numerous different file types. The contents of text files, for example, can be displayed on the screen using such commands as **cat**, **more** and **pg**. Doing this with a binary (or compiled program) file, may cause the screen to lock, as such files usually contain many control characters. (See "Looking at the contents of a file" (page 102) for more on the display commands and on garbling your screen.)

To avoid using an unsuitable command to display the contents of a file, first find out what kind of information a file contains. To do this, use the **file**(C) command, as follows:

```
$ file mbox
mbox:    ascii text
$ file tools
tools:   directory
$ file /bin/lc
/bin/lc:       iAPX 486 executable
```

The **file** command accepts either a simple filename or a pathname as an argument.

3

Looking at the contents of a file

The simplest way to look at the contents of a short file is to use the **cat**(C) command, as follows:

> **cat** *filename*

If the file is more than one screen long, it scrolls off the screen, making it difficult to read its contents. If this happens, press ⟨Ctrl⟩**S** to temporarily stop the scrolling, and ⟨Ctrl⟩**Q** to restart the scrolling. If you want to stop the scrolling completely, press ⟨Del⟩.

If you do not know what is in a file you want to look at, use the **cat -v** option, as follows:

> **cat -v** *filename*

This option causes any unprintable characters in *filename* to be displayed in a manner which does not garble your screen. If you do use **cat** without using the **-v** option, and your screen becomes garbled and the machine beeps a lot, press ⟨Ctrl⟩⟨Del⟩, ⟨Break⟩, or ⟨Del⟩ (depending on your terminal). If you cannot clear it, you may need to ask your system administrator for help.

To look at the contents of a file that is too big to fit on a single screen, use the **more** command, as follows:

> **more** *filename*

You can use the **pg** command in the same way.

You can look at more than one file at a time by using the display commands with several filenames as arguments, as follows:

> `$ `**`more file4 file5 file6`**

In the case of the **more** command, press ⟨Space⟩ to display a screenful of text. When you reach the end of the first file, **more** displays a message at the bottom of the screen (`Next file: filename2`). Press ⟨Space⟩ again to go to the next file.

If you want to go directly to the next file before finishing the first, enter **:n**; **more** skips to the next file. See "Listing the contents of a directory" (page 87) for more information on the **more** and **pg** commands.

Finding out how much text is in a file

The **wc**(C) command counts the number of lines, words, and characters in a file, using the options **-l, -w,** or **-c** respectively. For example, to print the number of characters and lines in a file called *myfile*, execute the following command:

```
$ wc -cl myfile
    32675  684  myfile
```

The order in which you specify the options determines the order of the output.

You can also give **wc** a list of files to count:

```
$ wc chap1 chap3
    105    676    3844 chap1
    675   3869   24269 chap3
    780   4545   28113 total
```

The *total* line gives sums for the lines, words and characters in the two files, *chap1* and *chap2*.

Looking at the beginning and end of a file

To look at the first ten lines of a file, use the **head**(C) command:

> **head** *filename*

To look at the last ten lines of a file, use the **tail**(C) command:

> **tail** *filename*

If you use a numerical option, for example **-20, head** or **tail** will print that number of lines (20) instead of ten, the default, as follows:

```
$ head -20 file6.txt
```

Copying a file

To copy one or more files, use the **cp**(C) command, which takes one of the following formats:

> **cp** *filename copyname*
> **cp** *filename ... pathname*

In the first format, *filename* (with optional path) is the name of the existing file that you want to be copied; *copyname* (with optional path) is the name you want the copy to be created with.

If you are using the second format to copy a group of files, you can only specify a directory, *pathname,* as the destination of the specified files. *filename* and *copyname* cannot be the same if they are both in the same directory.

When you copy a file you are creating a duplicate of it, which occupies additional space in the filesystem. Although the contents of the new file are the same as those of the original file, the new copy has its own inode number; any operations carried out on it have no impact on the original.

For example, to copy the file *project1* from your current directory to the directory */u/workgrp*, type the following command:

```
$ cp project1 /u/workgrp
```

The copy will retain the name *project1*, but will have a different pathname.

You can copy a file to your current directory by typing a command line like the following:

```
$ cp ../../a.out .
```

In this case, the file called *a.out* is located two levels above the current working directory, and is to be copied to the current location (as indicated by the "." notation).

> **NOTE** When copying a file, be careful not to overwrite an existing file. To avoid this, do not create a copy with the same name as an existing file, as this will overwrite (*clobber*) the contents of the existing file. This can be avoided by setting the **noclobber** variable. See "More about redirecting input and output" (page 256) for details.

When you copy a file, you automatically become its new owner. Accordingly, you must have read permission on a file before you can copy it. You can place files in any directory for which you have write permission. If you want to create a copy of a file without changing its ownership, use the command **copy -o** instead of **cp**; this preserves the owner and group of the file. For example, to copy */tmp/johnsfile* to your home directory without changing the ownership of the file, type the following:

```
$ copy -o /tmp/johnsfile johnsfile
```

For information on how you can assign the ownership of a file to someone else, see "Giving a file to someone else" (page 125) and "Access control for files and directories" (page 121).

Moving or renaming a file

To move one or more files to another directory, use the **mv**(C) command, as follows:

mv *filename ... pathname*

The one or more *filename* arguments (with optional path) specify the file or files you want to move; ***pathname*** is the path to the directory where you want to put the file.

For example, to move the file *project1* from your current directory to the directory */u/workgrp*, type the following:

```
$ mv project1 /u/workgrp
```

The procedure for moving files is the same as for renaming files. You rename a file by moving it to a new filename. To move (rename) a file, type the following:

mv *old_filename new_filename*

old_filename is the file's current name and *new_filename* is the name you want to change it to.

You can move a file to a different directory and rename it at the same time. For example, the following command line moves *chapter.1* to */u/workgrp* and renames it to *finished.chapter.one* at the same time:

```
$ mv chapter.1 /u/workgrp/finished.chapter.one
```

You can place files in any directory to which you have write permission. To move a file, you need read permission unless you own it.

> **NOTE** If you give a file the same name as an existing filename, the contents of the existing file are overwritten or "clobbered". The existing file is deleted. (You can make the system refuse to overwrite existing files by setting the **noclobber** variable: see "Specifying command input and output" (page 118) for details.)

Removing a file

To remove (or destroy) a file, use the **rm**(C) command, as follows:

rm *filename*

Once a file is removed from the system, there is no way of getting it back unless a backup exists on tape or floppy disk, or the filename is a *link*, or versioning is available. Links are explained in "Creating links to files and directories" (page 94); file versioning is explained in "Retrieving deleted files" (page 114).

You can list several files to be removed, or use wildcards to select files. You cannot remove directories with this form of the **rm** command.

> **NOTE** It is potentially dangerous to use wildcards to remove files. Before doing so, you should confirm that the correct files have been selected: do so by running the **ls** command in place of **rm**. Because the expansion of any filename notation is handled by the shell and not by the individual command, the files selected by **ls** are the same as those that will be selected by **rm**.

To remove files interactively, use the **-i** option, as follows:

> **rm -i** *filename1 filename2 ...*

rm with the **-i** option asks for confirmation before removing a file. A question mark is displayed and you can either type "y" to remove the file, or "n" to not remove it. It is a good idea to use **rm -i** to reduce the risk of accidentally removing files. For example, to remove several files from the current directory:

```
$ rm -i f*
file1: ?y
file2: ?y
file3: ?y
format.doc: ?n
```

As a further safeguard, it may be useful to create an alias, whereby executing **rm -r** * actually executes **rm -ir** *: the **-i** option causes **rm** to delete files *interactively*; that is, you must confirm the deletion of each file before it is carried out. See "Using aliases" (page 237) for details of how to create an alias.

Note that using wildcards does not remove hidden files (those whose name begins with a dot); that is, typing **rm** * does not necessarily remove all the files in a directory. To list the hidden files, type **ls -a**. For example, if you have a file called *.project*, you can remove it by typing the following:

> **$ rm .project**

Remember that there are always at least two files that cannot be removed from a directory; "." (the current directory), and ".." (the parent directory).

You can remove a file from a directory other than your current one if you have write permissions on that directory.

Removing files with difficult names

Occasionally, files are created by accident with awkward names. For example, they might contain a slash (/) or an asterisk (*) character. These files cannot be removed by normal means without the risk of destroying other files, because if you try to type their names, the shell will interpret the special characters as wildcards.

For example, suppose you have a directory that contains a corrupted file called *all* * *file* and a number of files called *file1* and *file2* that you want to keep. If you type **rm all** * **file**, **rm** will interpret the " * " in the filename as a wildcard, and attempt to execute the following:

> **rm all file1 file2 file**

This command thereby inadvertently deletes *file1* and *file2*.

To correctly remove files with corrupted names, the easiest solution is to use the **rm -i** option. In this case, **rm** will prompt you for confirmation before removing each of the specified files in the current directory; type "n" for each file other than the corrupt one you want to remove, as follows:

```
$ rm -i a*
all * file: ? y
```

Alternatively, specify the name of the file, surrounding it with single quotes:

```
$ rm 'all * file'
```

The single quotes prevent the shell from expanding the special character " * " in the file's name.

If you have a file that begins with a hyphen (-), **rm** will mistake its name for an option of some kind. For example, if your file is called *-myfile*, **rm -myfile** will be mistaken for an invalid **rm** command. You can overcome this by invoking **rm** with the special option, --, which tells **rm** that the following argument is not an option:

```
$ rm -- -myfile
```

See Chapter 12, "Regular expressions" (page 315) for an explanation of shell wildcards. See "Filenaming conventions" (page 83) for an explanation of what constitutes an illegal filename.

Comparing files

It is often necessary to compare the contents of two files and list any differences. This may be because you have made some changes to a file and cannot remember them; if you have a previous version of the file, you can compare the two. You may have two files with the same name in different directories; you can compare them to see if they are different files or two versions of the same file.

To see if two files differ, use the **cmp**(C) (compare) command which reads *file1* and *file2* and reports whether or not they are different:

cmp *file1 file2*

If they differ, **cmp** reports the point at which the two files diverge. This is reported in terms of the number of characters into *file1* at which the difference was detected, and the number of the line containing that character, as follows:

```
$ cmp chapter4 chapter4.bak
chapter4 chapter4.bak differ: char 28895, line 849
```

In this case, the two files are the same up to a point on line 849 of *chapter4*.

3

To see the precise differences on a line-by-line basis, use the **diff**(C) command, as follows:

diff *filename1 filename2*

For example, consider the following two files, *note.1* and *note.2*:

```
$ cat note.1
Charles
Please send me a report.
I need it for tomorrow's meeting.
Thanks
Bridget
$ cat note.2
Charles
Please send me a report today.
I need it for tomorrow's meeting.
Thank you
Bridget
```

To compare these files, line by line, use the **diff** command as follows:

```
$ diff note.1 note.2
 2c2
< Please send me a report.
---
> Please send me a report today.
4c4
< Thanks
---
> Thank you
```

The "2c2" means that there is a change ("c") between line 2 in the first file and line 2 in the second. Likewise, the "4c4" means that there is a change between line 4 in the first file and line 4 in the second. The "<" refers to a line in the first file, and ">" refers to a line in the second file. The "---" separates the output from each file.

If you want to compare three files, use the **diff3**(C) command. If you want to compare two sorted files, use **comm**(C).

Sorting the contents of a file

You can sort a file containing lines of text or numerical data in a variety of ways using the **sort**(C) command. For example, suppose you have a file called *names* containing the following:

```
perry
john
sarah
charles
```

To sort its contents alphabetically, enter the following command:

```
$ sort names
charles
john
perry
sarah
```

To direct the sorted output to a file (*names1*) rather than the screen (standard output), you can use either of the following command lines:

```
$ sort -o names1 names
$ sort names > names1
```

You can cause the original file to be sorted by giving the original filename for both arguments.

You can make **sort** merge two files together, in order. To do this, type the following:

> **sort** *filename1 filename2 > filename3*

This creates *filename3*, which contains the sorted, merged contents of *filename1* and *filename2*. (The **sort** command sorts the files as it merges them.) You can use the **-u** option to tell **sort** to make sure that each line in *filename3* is unique; that is, if both *filename1* and *filename2* contain an identical line, only one copy of the line will be written to *filename3*:

```
$ cat file1
perry
john
sarah
charles
$ cat file2
susanna
charles
bridget
john
```

Running the **sort** command on these files merges the contents and places them in alphabetic order, as follows:

```
$ sort -u file1 file2 >file3
$ cat file3
bridget
charles
john
perry
sarah
susanna
```

There are several more options that can be used with **sort**. For example, **-r** sorts in reverse order; **-n** sorts on numerical order, not text order; **-M** causes **sort** to assume that the first three characters of the field being sorted are months (like "JAN", "FEB", "MAR", and so on) and sorts them into date order.

You can make **sort** select any field in a line and have it base its comparisons on that field, as follows:

```
$ cat birthdays
charles FEB
bridget DEC
sarah   JAN
$ sort -M +1 birthdays
sarah   JAN
charles FEB
bridget DEC
```

The **+1** flag tells **sort** to make comparisons between records on the basis of the second field of each line. So, the month abbreviation on each line of the file is used as the basis for the sort operation above, and not the alphabetic order of the first field.

If you have a file where data records are made up of fields separated by some special character (called a "separator"), you can tell **sort** to use that separator by using the -t option, as follows:

```
$ cat birthdays
charles:FEB
bridget:DEC
sarah:JAN
$ sort -M +1 -t: birthdays
sarah:JAN
charles:FEB
bridget:DEC
```

Searching for text in a file

To search one or more files for some text, you can use the **grep**(C) command, as follows:

> **grep** *options text filenames*

("grep" is an acronym for "global regular expression print"; for a full explanation of regular expressions, see Chapter 12, "Regular expressions" (page 315).)

grep searches the contents of *filenames* for *text*, and prints any matches. You might want to do this if you cannot remember the name of a file in which you left some information, but can remember enough of it for **grep** to find it for you.

For example, you might want to locate a memo in the current directory (full of files called *something.memo*), when you know that the file you are looking for contains the string "Subject". The command to use is as follows:

```
$ grep 'Subject' *.memo
stan.memo:Subject:  That's another fine mess you've gotten us into!
```

grep prints the context of any matches, line by line, with the relevant filename (where more than one file was specified for the search) followed by the line of text that contains the specified string.

The single quote (' ') marks are necessary if you want to search for a string containing spaces, tab characters, or double quote marks. Double quote (" ") marks are necessary if you want to search for a string containing single quote marks; you should put a backslash immediately in front of each quote character (\' \'), as follows:

```
$ grep "I\'m right" stan.memo
Thanks for nothing. I'm right in the center of it (or
```

If you are not sure whether the string is uppercase, capitalized, or all lowercase letters, use the **grep -i** (ignore case) option; **grep** ignores the case of the text in the files being searched, and report all matches, as follows:

```
$ grep -i 'PhD' database.memo
yesterday, when he was awarded his PhD in New
not so easy to get a pHD nowadays, what with
is it PHD, phd, phD, etc? He should have stopped
```

This search has found all lines containing the string "PhD", irrespective of how it is capitalized.

If you want to see all lines in a file that do *not* contain the string, use **grep -v**.

The use of regular expressions and pattern matching in search operations is explained in Chapter 12, "Regular expressions" (page 315). See also **regexp**(M).

If you have a file containing columns of data in textual form, you can extract information from it using a variety of tools. For example, supposing you have a file called *blackbook* containing names, extension numbers, login names and dates, in a format like the following:

```
Michael Stand:571:mikes:JAN-1-91
Sue Penny:284:suep:FEB-6-89
Joshua Ford:954:joshf:JUL-30-88
Liz Addams:553:liza:AUG-15-91
```

To see Sue Penny's record, use the following command:

```
$ grep Sue blackbook
Sue Penny:284:suep:FEB-6-89
```

This is hard to read. To see only Sue's extension number (the second field), you can use the **cut**(C) command, as follows:

```
$ grep Sue blackbook | cut -f2 -d:
284
```

The **cut** command extracts individual fields from a file containing records. The **-f2** option tells **cut** to extract only the second field of each record; the **-d:** option means that fields are delimited with a colon. In this way, input records may contain spaces and tabs without these characters signaling the start of a new field.

The *pipe* (|) tells **grep** to send its output to another program (in this case, to **cut**) instead of the standard output. See "Running commands in a pipeline" (page 120) for more information on pipes.

To see a list of all the people in your file, followed by their login names, you do not need to use **grep**: instead, use the **cut** command, as follows:

```
$ cut -f1,3 -d: blackbook
```

The **-f1,3** option tells **cut** to extract the first and third fields in each record:

```
Michael Stand:mikes
Sue Penny:suep
Joshua Ford:joshf
Liz Addams:liza
```

If you want to put your list in alphabetic order, you can sort it as follows:

```
$ cut -f1,3 -d: blackbook | sort -df
Joshua Ford:joshf
Liz Addams:liza
Michael Stand:mikes
Sue Penny:suep
```

A more powerful and versatile tool for this sort of operation is the **awk**(C) command. See Chapter 13, "Using awk" (page 323) for an explanation of its use.

Permanent executable copies of complex command lines like these search tools can be stored in shell script files for future use. See Chapter 11, "Automating frequent tasks" (page 245) for details.

Finding files

To search the system for a particular file, use the **find**(C) command, as follows:

find *start_point* **-follow -name** *filename* **-print**

The *start_point* argument tells **find** where to start searching in the filesystem, for example, *root*. **find** searches its starting directory, and all the subdirectories. If you know your file is in one of your own subdirectories you could tell **find** to start searching from **$HOME**.) The **-follow** option tells **find** that if it encounters a symbolic link, it should follow it to the file the link points to, as described in "Navigating symbolic links" (page 97). The **-name** option is followed by the name of the file you are looking for. Every time **find** sees a file with this name, it carries out the actions specified by the subsequent options. For example, the **-print** option tells **find** that the action it must take when it finds *filename* is to print its pathname:

```
$ find /tmp -name myfile.tmp -print
/tmp/myfile.tmp
```

find gives lots of error messages when you do not have permission to search a directory, for example:

```
$ find / -name chap3 -print
/u/charles/stuff/os/chap3
/u/w/Xenix/OS2.3.2/Intlsupp/Guide/chap3

find: cannot chdir to /etc/conf/pack.d/arp
find: cannot chdir to /etc/conf/pack.d/arpproc
    .
    .
    .
```

To suppress these error messages, redirect the error output of the **find** command to */dev/null* (the UNIX system's "black hole" directory), as follows:

```
$ find / -follow -name chap3 -print 2> /dev/null
```

For more information on redirecting output, see "Specifying command input and output" (page 118).

find can be used to apply a command to a collection of files that match some selection test, for example, files that are older than a specified age. You can remove all files in your home directory, and all its subdirectories, that have not been accessed for seven days by typing the following command line:

```
$ find $HOME -follow -name '*' -atime +7 -exec rm {} \;
```

find starts from the directory specified by its first parameter (in this case, the value of **$HOME**), follows symbolic links, and selects all the files matching the designated name (in this case, '*') that were last accessed (**-atime**) seven or more days ago. It then executes (**-exec**) the **rm** command on the found file (represented in the expression by {}). The "\;" at the end of the line terminates the **-exec** expression. Note that the single quotes around the " * " are required. Otherwise **find** searches for files with names matching those matched by " * " in the current directory.

The **-exec** option allows the execution of any legal shell command along with any permitted options and arguments.

find can carry out other tasks when it finds a file. For example, the following command causes **find** to execute **cp** on any file called *datafile* in the directory */bin*; this file is then copied to your home directory.

```
$ find /bin -follow -name datafile -exec cp {} $HOME \;
```

Retrieving deleted files

File versioning is the ability of a system to preserve and access old copies of a file. Traditionally, the UNIX system does not support file versioning: whenever the UNIX system updates a file, the preceding image of its contents is lost. Certain editors make a copy of a file before updating it, but this feature is specific to the individual utility, and usually restricts itself to the maintenance of the current file and a single backup version.

Keeping old versions of files

The SCO OpenServer system supports file versioning. The root user must configure it for a filesystem at the time of mounting by setting the **MINVTIME** and **MAXVDEPTH** kernel parameters. These respectively set the interval before a file is versioned and set the maximum number of versions maintained (or steps in the evolution of the file's contents). Setting these to 0 disables versioning. (See *System Administration Guide* for details.)

When file versioning is configured, you must then enable it for a specific directory using the **-s** option of the **undelete**(C) command, as follows:

```
$ undelete -s /users/mike/source/user_guide
```

This command line switches on versioning for all the files in the directory called *user_guide* and any subsequent child directories.

When file versioning is configured and enabled, it makes the filesystem preserve a copy of a file's contents whenever it is overwritten or deleted. This is done silently. The number of versions preserved depends on the setting of **MAXVDEPTH**. These copies can be retrieved at a later date using **undelete**.

You can make versioning visible by setting the **SHOWVERSIONS** environment variable to 1, as follows:

```
$ SHOWVERSIONS=1; export SHOWVERSIONS
```

Within a directory, you can create versions for a single file, irrespective of whether general versioning configuration or enabling has been carried out, using the **undelete -v** option, as follows:

```
$ undelete -v chapter3
```

> **NOTE** File versions created in this way will always be visible, independently of the value of **SHOWVERSIONS**. However, when the filesystem is mounted with versioning enabled, file versions created using **undelete -v** will not be visible.

A versioned filename has the following syntax:

filename;version

The *filename* element follows the normal UNIX file naming rules, and is separated from the file version by a semicolon.

To see a list of all the existing versions of a file, use the **undelete -l** option, as follows:

```
$ undelete -l magnolia.txt
: no versions
$ undelete -l begonia.txt
begonia.txt;1
begonia.txt;2
begonia.txt;3
```

Specifying a version identifier causes only that version to be used in the command line; the existing versions are accessible in the same way as separate normal files:

```
$ cat homework.txt\;1
Italian Assignment 3

C'era una volta, l'orsetto che viveva nella foresta al ovest
. . .
$ cat homework.txt\;2
Italian Assignment 3

Tanti anni fa, l'orsetto che vivo' nella bosca verso ponente
. . .
```

Note that the semicolon is a shell metacharacter, and must be quoted in the context of versioning, using the backslash. See Chapter 12, "Regular expressions" (page 315) for more information on metacharacters and quoting.

Undeleting files

On traditional UNIX systems, once you have deleted a file, you cannot retrieve it, other than by searching through any existing backup tapes. The SCO Open-Server system **undelete** command makes this process much easier on versioned files. These exist in three combinations:

- An existing file with no previous versions.

- An existing file with one or more previous versions.

- A file that no longer exists but which has one or more previous versions.

To undelete a file that was removed by mistake, make a previous version current. Although the file itself may have been removed, the versions are still accessible, as the following sequence shows:

```
$ l
-rw-r--r--   1 sallyp    accounts    3768 Oct 31 16:03 begonia.txt
$ rm begonia.txt
$ l
total 0
$ undelete begonia.txt
$ l
-rw-r--r--   1 sallyp    accounts    3768 Oct 31 16:04 begonia.txt
```

When you specify a filename without a version identifier, you retrieve the most recent version of the file. However, it is possible to check how many versions are available, using the **undelete -l** option, as follows:

```
$ undelete -l
begonia.txt;1
begonia.txt;2
begonia.txt;3
$ undelete -i begonia.txt
begonia.txt;1: ? n
begonia.txt;2: ? n
begonia.txt;3: ? y
$ l
-rw-r--r--   1 sallyp   accounts    3768 Oct 31 16:05 begonia.txt
```

The **undelete -i** option interactively recovers any available versions of the specified file. In this case, the most recent version (number 3) of the file is retrieved but the earlier versions are discarded.

```
$ undelete -l
: no versions
```

Cleaning up your filesystem

File versioning is a useful tool, but it can have an adverse effect on the filesystem, as unneeded old versions of files can rapidly accumulate, reducing free space. It is important to remember that switching on versioning for a directory causes it to apply to all child directories as well. Therefore, directories should be regularly cleared of unwanted old file versions.

The **undelete -p** option "purges" specified files. This permanently deletes the existing versions (but not the file itself).

```
$ undelete -l
tulip;1
tulip;2
tulip;3
tulip;4
$ undelete -p tulip
$ undelete -l
: no versions
```

Versioning is inherited by child directories, so the following command line is useful for cleaning out a whole directory system:

```
$ undelete -prd /users/mike/source/user_guide
```

The **-r** option specifies a recursive purge, while **-d** specifies the directory at which the operation is to commence.

Another useful option is **-m**, which takes a number of days as an argument.

```
$ undelete -m+2
$ undelete -m-2
$ undelete -m2
```

These command lines respectively cause **undelete** to consider only files deleted more than, less than and exactly, two days ago.

In filesystems where versioning is widespread, the issue of free space may be crucial. Therefore, it is advisable to set up a **crontab**(C) job containing a command line like the following, which executes a forced recursive purge of all files deleted more than two days ago, starting at the directory /*users*:

```
$ undelete -rpfm+2 -d /users
```

See "Executing processes at regular intervals" (page 170) for details of the **crontab** command.

Specifying command input and output

Almost all UNIX commands require an *input* and an *output*; that is, some information to read and process, and somewhere to store the results. If you do not tell a command where to find its input and output, it makes assumptions about where to read and write information. These assumptions are called the *standard input* and *standard output*. These are, respectively, your keyboard and your screen by default. Alternatively, if you specify the name of a file, most programs will obtain their standard input by reading the file.

In addition to standard input and standard output, most programs need a *standard error*, to which they report any failures or errors. By default, the standard error is directed to the same place as the standard output, your screen.

You can make commands redirect their standard input and output by using the symbols " < " and " > " on the command line, followed by the name of a file to read input from or write output to. For example **sort < list1 > list.out** makes **sort** treat *list.1* as its input, and send its output to *list.out*.

If you send the output of a program to a file, and the file already exists, the existing file will be "clobbered" or overwritten. If you are using the Korn shell, you can prevent this from happening by using the **noclobber** variable. You can identify your login shell by entering the following command:

```
$ grep ${LOGNAME} /etc/passwd
martins:x:13990:1014:Martin Smith:/u/martins:/bin/ksh
```

The last data field, after the last colon, identifies your login shell, in this case, the Korn shell (*/bin/ksh*). If you are using the Korn shell, you can turn on the **noclobber** feature, by typing the following:

```
$ set -o noclobber
```

C shell users should type **set noclobber**: this feature does not exist in the Bourne shell. To turn off the feature in the Korn shell, type **set +o noclobber**. C shell users should type **unset noclobber**. To protect yourself from accidentally clobbering files, Korn shell users should add the appropriate **noclobber** line to your *.profile* file: C shell users should add it to *.cshrc*.

To append the output of a command onto the end of a file, use the >> notation instead of >. For example, the following command line appends the output from **sort** to the end of the existing contents of *file2* rather than overwriting them:

```
$ sort file1 >> file2
```

The C shell will not let you append the standard output to a file if the file does not exist and **noclobber** is set. The Bourne or Korn shells simply create the file.

Typically, your shell will support a wider range of redirection operators than those discussed here. For details, refer to **ksh**(C), **sh**(C) or **csh**(C) as appropriate. See also "More about redirecting input and output" (page 256).

Forcing a program to read standard input and output

Many programs get their input from the standard input and write their output to the standard output: others read from or write to named files.

For example, **cat** can be used to create a file using information typed in at the keyboard, as follows:

```
$ cat > file8
```

In this case, *file8* is a file that does not already exist within the current directory. You can then proceed to type text into the file. You can press ⟨Bksp⟩ to correct any mistakes you make on the current line, although you cannot correct mistakes on previous lines. Press ⟨Ctrl⟩D when you have finished, to signal "end of file" to **cat**.

Alternatively, you can use the **echo** command to place text in a file, as follows:

```
$ echo "Hello there!" > testfile
```

This results in a file (*testfile*) containing the text "Hello there!".

Note that the quote marks are stripped out by the shell when you use **echo** to print to a file.

The following use of the **cat** command places the contents of the three named files into *outfile*:

```
$ cat file1 file2 file3 >outfile
```

Suppose you want **cat** to read *file1*, then read something from the standard input (your terminal) instead of from *file2*, then read *file3*. There exist some special device files to make life easier: */dev/stdin*, */dev/stdout*, and */dev/stderr*. These three special files correspond to the standard input, standard output, and standard error, respectively. The following command line uses */dev/stdin* in precisely this way:

```
$ cat file1 /dev/stdin file3 >outfile
```

In this case, *file1* is copied to *outfile*; then **cat** reads the standard input (your terminal) until you press ⟨Ctrl⟩**D** (to signal end of file); then finally appends *file3* to *outfile*.

Running a sequence of commands

When you type a command line and press ⟨Enter⟩, the entire line is evaluated as a single unit. It is possible to run several programs from one line; either sequentially, or simultaneously, or in a "pipeline" where the output from one command is used as input to the second command. You can also store frequently-used commands in a file, and tell the shell to execute the contents of the file as a script.

Entering commands on the same line

To send several commands, one after another, separate each of them with a semicolon. For example:

```
$ ls > list ; sort list > list1
```

This command sequence creates a list of files in a file called *list*. It then sorts the contents of the file alphabetically and redirects the output into a file called *list1*. The command after the semicolon is not executed until the command before it has completed; the shell waits for the earlier commands to finish.

Running commands in a pipeline

A pipeline is a sequence of commands that operate concurrently on a stream of data. All the processes are started simultaneously, but instead of reading or writing to a file or terminal, they read or write to and from a *pipe*. As the first process begins to produce some output, that output is fed to the second process as input, so that both processes are working at the same time. For example:

```
$ ls | sort > list1
```

Here, the **ls** command sends its output straight to **sort**, which processes it and sends its own output to the file *list1*. Unlike the similar command line in "Entering commands on the same line" (page 120), no intermediate file called *list* is created. Writing to a temporary file is a comparatively slow process because it involves transferring data to disk, and the second process must then access the file and read it back into memory. Pipes, in contrast, transfer data directly from one process to another without writing it to the disk.

More than one pipe operation can appear on a single command line, as follows:

```
$ sort -u file | grep basilisk | wc -l > words
```

This pipe sequence creates a file called *words*, containing a count of all the nonidentical lines in *file* that contain the word "basilisk".

Access control for files and directories

Because the SCO OpenServer system is a multiuser system, it is important that strict control is placed on file access. For example, as a user you cannot change files that belong to someone else without their authorization. Controlling access to files is achieved by use of *permissions*.

Every file has three sets of *permissions* that control who can read it, write it (that is, change it), and execute it. You can change the permissions on your own files to make them more or less accessible to other users on the system. The following is a representation of the permissions information displayed by the **ls -l** command. Remember that the first character position actually gives the file type, and is not a permissions indicator; see "File and directory attributes" (page 81):
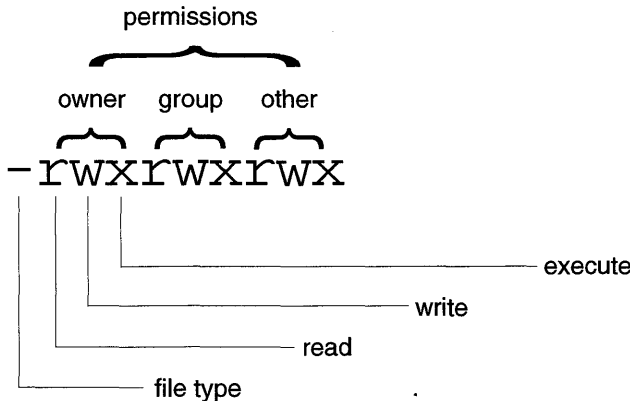
The permissions field for a file is made up of nine character positions following the file type indicator. They are divided into three sets of three permissions each; a set for the owner of the file, a set for the group of users to which the file belongs, and a set for everyone else on the system. These are respectively known as "owner", "group" and "other".

Note that the superuser (root) can always read or write every file on the system. This is a special privilege that is not available to any other user.

Each set of permissions can include none, one, or more than one of the following privileges:

Read   If you have *read* permission, you can look at the contents of a file. For a directory, this means you can see a list of the files it holds. Read permission is represented by an "r" in the first of the three character positions for each of the three sets of permissions, as follows:

```
-r--r-----   1 johnd   unixdoc    10586 Feb 25 12:26 1.start
```

The "r" in the first character position of owner's set and the group set means that the owner and members of the owner's group can read the file; nobody else is permitted to do so.

Write   If you have *write* permission on a file, you can alter its contents. For a directory, this means you can create files and subdirectories within that directory. It also means you can remove files from that directory even if you do not have write permission on the files.

```
--w--w--w-   1 johnd   unixdoc     8660 Feb 25 13:08 2.start
```

The "w" in the owner's set, the group set and the other users' set means that all classes of user can alter this file.

You cannot remove a file unless you have write permission on the directory it is stored in. If you try to remove a file from a directory for which you do not have write permission, you will see an error message like the following:

```
$ rm freds.file
rm: fred/freds.file not removed.
Permission denied
```

Execute For a file, this means that if the file is a program, you can execute it. Execute permission on a directory means you can change to it.

```
---x--x--x   1 johnd   unixdoc        Feb 25 13:08 2.start
```

In all cases, a hyphen in any of the permissions fields indicates that the permission is not set.

More uncommonly, you may encounter other permissions in a long listing, for example "s" or "t". For details, see **ls**(C).

To see the permissions on the current directory, use the **l -d** (directory) command, as follows:

```
$ l -d
drwxrwxrwx  21 johnd    techpubs    1552 Dec 07 15:40 .
```

Changing file permissions

To change the permissions on a file, use the **chmod**(C) (change mode) command, which has two formats, "symbolic" and "absolute", as follows:

> **chmod** *who operator permission filename*
> **chmod** *mode filename*

Using the first, symbolic, format, the *who* field is one or more of the following characters:

a   all users; change all three sets of permissions at once

u   user; change the user, or owner, permissions

g   group; change the group members' permissions

o   others; change the other users' permissions

The *operator* field is one of the following symbols:

+   add a new permission

-   remove a new permission

=   set permissions while clearing (removing) all other permissions

The following sample usages of **chmod** show a number of symbolic permissions being set:

$ **chmod g+w memo**   adds write permission for group members on the file *memo*.

$ **chmod o-wx memo**   removes write and execute permission for others (users other than the owner or those in the file's group).

$ **chmod o= memo**   clears (removes) all permissions for other (setting a NULL permission clears any existing value).

$ **chmod u=rx memo**   sets read and execute for user, clearing (removing) write permission (which is not specified in the "=" command.)

$ **chmod a+w memo**   adds write permission to the existing permissions for all categories of user.

3

You can also change permissions using their absolute numeric values, by giving a three-digit octal number to specify the permissions. This method is harder to use but less verbose.

Using octal numbers to set permissions

Permissions	Octal number
---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

Permission to execute a file is represented by a value of 1. Permission to write a file is represented by a value of 2. Permission to read a file is represented by a value of 4. These values are added together to produce the combinations in the table above.

Three octal numbers (numbers in the range 0 to 7) are used to represent the owner, group and other permissions respectively. Thus, by adding the permissions for a given category of user, you produce a digit; and by specifying three digits (one for each set of users) you can specify all the permissions on a file, as follows:

```
$ l myfile
-rw-r--r--   1 johnd techpubs    5061 Feb 10 15:01 myfile
$ chmod 640 myfile
$ l myfile
-rw-r-----   1 johnd techpubs    5061 Feb 10 15:01 myfile
```

myfile originally possessed permissions 644. The "6" gives read and write permissions (2 plus 4) to users in the specified group, while the "4" gives read permissions only. "0" gives no permissions at all. The effect of executing **chmod 640** on this file was to deny all permissions to users of group "other".

Setting the default permissions for a new file

When new files are created, their initial permissions are determined by their *file creation mask*. The **umask**(C) command is executed whenever you log in, and it automatically sets the mask to restrict the permissions placed on any files that you create. You can change the permissions placed on new files by running **umask** again; the new permissions override the old ones.

To change the permissions applied to a newly created file, specify the permissions you want to have *removed* from the new file. In this way, specifying a file creation mask of **o=rwx** causes read, write and execute permission to be *denied* to other users.

```
$ touch test
$ l test
-rw-rw-r--   1 charles techpubs       0 Feb 22 09:29 test
$ umask u=,g=w,o=rwx
$ touch test.2
$ l test.2
-rw-r-----   1 charles techpubs       0 Feb 22 09:30 test.2
```

The **touch**(C) command creates an empty file, in this case called *test*.

In the command lines above, the **umask** command specifies that write permission is to be removed from members of the file's group, and that read, write, and execute permissions are to be removed from other users. No change is made to the permissions available to the file's owner.

> **NOTE** Where the = operator is used in **umask**, it has the opposite effect to the = in **chmod**. With **chmod**, it sets any specified permissions, and unsets the rest, whereas with **umask**, it unsets the specified permissions while setting all the others.

Note that you cannot normally create an executable file using **umask**; you can only change a file's permissions to make it executable. For example, if your **umask** is **umask u=,g=,o=rwx** this gives your default file permissions of 660 (rw-rw----), not 770 (rwxrwx---), even though execute permissions for user and group have not been removed. The only exceptions to this rule are when creating a directory or compiling a program to create an executable binary (in which case the executable bits are set in accordance with your **umask**).

You can set **umask** using octal permissions. To set the **umask**, work out what permissions you want to give newly created files in octal, then subtract them from 777. (Remember, the permissions specified in your **umask** are removed from the file, not added.) Accordingly, **umask 022** removes write permission from the group and other user classes: a file created with an initial mode of 777 becomes 755 and a file created with 666 becomes 644.

Giving a file to someone else

To give a file to someone else, change the ownership of the file with the **chown**(C) (change owner) command, as follows:

> **chown** *new_owner filename*

The *new_owner* argument is the login name of the new owner.

For example, the following command line assigns ownership of *01.intro* to the user *charles*:

```
$ chown charles 01.intro
```

You must be the current owner of a file to change its ownership; that is, you cannot give the file to someone else unless it is yours to give. When you create a file, you automatically become its owner.

Depending on the permissions on a file, if you give away ownership you may give away your right to access the file afterwards.

Finding out your group

In order to find out the groups of which you are a member, use the **id**(C) command, as follows:

```
$ id
uid=13052(johnd) gid=1014(techpubs)
```

The command displays your numeric user identification (UID) and your group identification (GID). Your login and group names are given in parentheses.

Changing your current group

Group control is carried out using the **sg**(C) (supplementary group) command. Type **id** (see "Finding out your group" (this page)) or **sg** to obtain a list of the groups of which you are a member, as follows:

```
$ sg
Current effective supplemental groups:
1014(techpubs)
```

You can change your current group by using the **sg -g** option, as follows:

```
$ sg -g techpubs
```

You must be recognized as a member of the new group before you can switch to it. Group memberships are listed in the file */etc/group*; each group has a line in the file, followed by the names of those users who are authorized to work in it. After successfully changing group, you work within the new group for the remainder of the login session (or until you run **sg -g** again).

Changing the group of a file

To change the group of a file, use the **chgrp**(C) (change group) command, as follows:

chgrp *new_group filename*

For example, to change the group of a file called *using_unix* to *techpubs*, use the following command:

```
$ chgrp techpubs using_unix
```

Files and users on the system are identified as members of a group by their group name. Groups, together with group permissions, allow people who need to use the same files to share those files without sharing them with all users. When you create a file, it is automatically given the same group as your own. You must be the owner of a file to change its group.

Printing a file

When you issue a print command, a copy of the file to be printed is *spooled*. It then waits in the print queue, along with other print jobs, until its turn comes to be printed. Because the system spools print jobs, you can go straight on to another task.

To print a file use the **lp**(C) (line printer) command. For example:

```
$ lp myfile
request id is laserwriter-635 (1 file)
```

This command sends *myfile* to the print queue. The "request id" line means that the file will be printed on the printer named "laserwriter", and is request number 635.

To print several files, add them to the command line, as follows:

```
$ lp file1 file2 file3
```

This command line prints *file1*, followed by *file2*, followed by *file3*.

Note that it is a bad idea to print executable programs, or other files containing binary data; in general you should only print files containing text, or containing some form of data intended for printing (such as PostScript® files).

By default, files are printed in *portrait* orientation on the paper: to print a file in *landscape* orientation (that is, sideways, so that long lines fit on the page), use the following command:

```
$ lp -ol file1
```

To print a PostScript file on a PostScript printer, you should specify that the file contains PostScript, by using the following command:

```
$ lp -og
```

See **lp**(C) for more information about how to send files to the printer.

If you need to add page numbers to a long file, use the **pr**(C) command, which prints files to its standard output, separated into pages with a header containing the page number and date and time of printing. You can then pipe the paginated output to **lp**.

For example, to print */etc/profile* in this way, use the following command line:

```
$ pr /etc/profile | lp
```

Printing several copies of a file

To print several copies of a file, use the **lp -n** option, as follows:

```
$ lp -n 3 file1
```

The argument to **-n** is the number of copies you want, in this case, three. The default number is 1.

Selecting a printer

If you know that several printers are connected to your system, and you want to send a file to a printer that is not busy, you need to know the destination printer's name. You can get a list of the printers available to you by using the **lpstat**(C) (line printer status) command, as follows:

```
$ lpstat -s
```

To select one of the available printers, use the **lp -d** option, as follows:

```
$ lp -dlaserwriter2 file1 file2 file3
```

This command line sends the specified files to the named printer.

You can assign a default printer for **lp** to use, by setting the **LPDEST** environment variable. Environment variables are explained in depth in "Setting shell variables" (page 227). Add a line setting the value of **LPDEST** to the name of your default printer to the appropriate login script. (Login scripts are described in "What happens when you log in" (page 224).)

For example, if you are using the Korn shell and the printer you want to use by default is called **postscript-2**, you can add the following line to your *.profile* file:

```
LPDEST=postscript-2; export LPDEST
```

postscript-2 will then become your default printer next time you log in.

Displaying a list of current print jobs

To display the list of current print jobs, use the following command line:

```
$ lpstat -o
```

lpstat reports on printer status.

Canceling a print request

To cancel a print request, use the **cancel**(C) command. You need to know the print request number that was assigned when the request was first spooled. This can be found using **lpstat -o**. For example:

```
$ cancel laserwriter-635
```

You can only cancel your own print requests.

Getting help on the command line

Extensive online help is provided by the **man**(C) system. **man** is short for "manual", and is a tool for retrieving the text of the reference manuals.

In this book, you will sometimes see reference keywords followed by a letter in brackets; for example, **ls**(C) or **regexp**(M). The letter in brackets indicates the section of the reference manual in which the keyword is discussed. If you are working at the shell prompt, you can read the reference entry for the keyword by entering the following command:

man [*section*] *keyword*

The *section* field is optional and is used to select a particular section when a keyword is documented in more than one section. For example, type **man C kill** to read the C section entry on the keyword **kill** (which is documented as a command in section C and a callable function in section S).

The reference manual entries are technical descriptions; they are not tutorials and make no concessions to the inexperienced user.

Getting help when you are uncertain of the topic

If you know the keyword but do not want to read all the reference text, you can use the **whatis**(C) command to list the description of the item. For example, to read the description of **man**, type the following:

```
$ whatis man
man(C)           - prints reference pages in this guide
```

If you are not sure of the keyword to use for a topic, you can use the **apropos**(C) command (which is the same as **man -k**). Each entry in the reference manual has a description associated with it; **apropos** searches the descriptions for the word you give as a subject. For example, to find reference entries concerned with searching, type **apropos search**. The following entries are among those displayed:

```
egrep(C)                  - Search a file for one or more patterns
fgrep(C)                  - Search a file for a fixed string
grep (C)                  - Search a file for a pattern
```

You can then use **man C grep**, for example, to display the manual page on the **grep** command.

Chapter 4

Editing files

The SCO OpenServer system has several editors, which are useful for different purposes:

vi Visual editor. **vi** allows you to perform full-screen writing and editing of files; this is the editor you will use most.

view Read-only version of **vi**. **view** allows you to examine text files, but does not allow you to save changes. See **vi**(C).

ed Original UNIX system line editor. **ed** is *line-oriented*; it can only edit a line at a time. Used within shell scripts, and when it is impossible to configure a terminal properly. See "Using ed" (page 154) for details.

ex A line editor. Like **ed**, **ex** is line-oriented. Like **view**, **ex** is implemented as part of **vi**, and the **ex** command is a link to **vi**. See **vi**(C).

sed Stream editor. **sed** reads an input file, carries out a sequence of commands, and writes the result to its output file. It cannot be used interactively. (See Chapter 14, "Manipulating text with sed" (page 371) for further details.)

4

A quick tour of vi

vi is the standard UNIX system tool for editing text. There are a few simple commands that you need to learn before you can use **vi**. These commands allow you to:

- start **vi**

- insert text

- search for text

- move around inside a file

- delete text

- save your changes and quit **vi**

The following example uses **vi** to enter and correct the first five lines of the poem "Kubla Khan" by Samuel Taylor Coleridge. Commands are shown at the left-hand side and described at the right:

vi kubla_khan

Start editing a new file called *kubla_khan*. **vi** displays a nearly empty screen with the cursor in the top left-hand corner. See "Starting vi" (page 134) for more details about the options available when starting **vi**.

i

Switch to insertion mode so that you can start to enter text. See "Entering text" (page 135) for more information about commands that you can use to enter text.

Type the following lines of text, pressing ⟨Enter⟩ to go to a new line. If you type a wrong letter, either leave it for correction later or press ⟨Bksp⟩ to move the cursor over it and enter the correct character:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Beta, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

⟨Esc⟩   Return to command mode.

/Beta⟨Enter⟩

Search for the word "Beta". The cursor is placed under the "B". See "Searching for text" (page 139) for more information.

xxxx   Delete the word "Beta". Each time you press "x", **vi** deletes the character under the cursor. See "Deleting and restoring text" (page 138) for details of other commands that remove text. "Using buffers to cut and paste text" (page 146) describes more sophisticated techniques that you can use to move blocks of text in **vi**.

iAlph⟨Esc⟩ Insert the word "Alph" and return to command mode. If you make a mistake while entering "Alph", press ⟨Esc⟩ to return to command mode and press **u** to undo the change.
Pressing **u** again undoes the undo. You should also refer to "Replacing and modifying text" (page 140) for information about replacing a single text string, and "Substituting text" (page 141) for details of how to search for and substitute several instances of a text string.

:w⟨Enter⟩ Save the changes you have made to the file *kubla_khan*. You should do this fairly frequently while you are editing to protect your work against accidental loss. See "Saving files and quitting vi" (page 136) for more information about saving to files.

k   Move to the previous line. This is one of the cursor movement commands available in command mode. The effect of pressing "k" is immediate; you do not need to press ⟨Enter⟩. See "Moving around a file" (page 137) for more details.

The edited file should now look like the following:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

:x⟨Enter⟩ Save the changes to *kubla_khan* and quit **vi**.

vi provides many more commands, shortcuts, and other features that you can use. The following sections cover these aspects of **vi** in greater detail.

4

Starting vi

To edit a file, type **vi** *filename* and if the file already exists, **vi** will read it in. If it does not exist, **vi** will create it. (See "Filenaming conventions" (page 83) for information about naming files.) In this example, the command **vi soliloquy** loads the file *soliloquy* in the current directory. You may notice that this version of Hamlet's soliloquy contains some misquotations; these will be removed in later sections:

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them. To die, to sleep -
No more - and by a sleep to say we end
The heartache, and the thousand natural shocks
Which flesh is heir to! 'Tis a consummation
Devoutly to be wished. To die, to sleep -
To sleep - perchance to dream: ay, there's the rub,
For in that sleep of death what dreams may come
When we have shuffled off this mortal coil,
Must give us pause. There's the respect
Which makes calamity of so long life:
For who would bear the whips and scorns of time,
Th' oppressor's wrong, the proud man's contumely,
The pangs of despised love, the law's delay,
The insolence of office, and the spurns
Which patient merit of th' unworthy takes,
When he himself might his quietus make
With a bare needle? Who would needles bear,
To grunt and sweat under a weary life,
          .
          .
          .

The fair Ophelia! - Nymph, in thy orisons
Be all my sins remembered.
"soliloquy" 35 lines, 1502 characters
```

On the bottom line of the screen, **vi** reports *soliloquy* as having 1502 characters on 35 lines of text:

```
"soliloquy" [Read only] 35 lines, 1502 characters
```

This line indicates that the file permissions on *soliloquy* are set so that you may not write to it. If you own this file, and you wish to make changes to it, change its permissions from within **vi** and reload it for editing using the following commands:

:!chmod u+w soliloquy
:rew

When you start **vi** you are in command mode. **vi** has two modes; command mode and insertion mode. In command mode you can issue commands to **vi** and move around your document. In insertion mode, you can only enter text.

If you want to start editing a file at a point part of the way through it, rather than at the beginning, you can start **vi** with the following command:

> **vi** +*line filename*

where *line* is the line number to position the cursor at. To start editing at the end of the file (for example, to append information to a list), start **vi** with the following command:

> **vi** + *filename*

Entering text

You must be in command mode before you can issue a command to enter text; while you are entering text, you cannot issue any command except ⟨Esc⟩ to return to command mode. If you would like a visible reminder when you are in insertion mode, enter the command **:set showmode** (see "Configuring vi" (page 152) for more details). The following commands put you in insertion mode to enter text:

a	Add text to the right of the cursor.
A	Add text to the end of a line.
i	Insert text to the left of the cursor.
I	Insert text at the start of a line.
o	Open a new line to put text below the current line.
Go	Go to end of the file and open a new line.
O	Open a new line above the current line.
1GO	Go to the first line in the file and enter new text above the line.

When you finish entering text, press ⟨Esc⟩ to return to command mode.

Taking the previous example, entering **Go** moves you to the end of the file and opens a new line. Pressing ⟨Enter⟩ adds a blank line. Typing "by William Shakespear" and pressing ⟨Esc⟩ adds new text. The end of the file *soliloquy* now reads as follows:

```
The fair Ophelia! - Nymph, in thy orisons
Be all my sins remembered.

by William Shakespeare
```

What to do if you get stuck

First press ⟨Esc⟩ twice. If a command is in progress, the ⟨Esc⟩ key cancels it. If you are in insertion mode, the ⟨Esc⟩ key puts you back into command mode. If your terminal beeps or flashes when you press ⟨Esc⟩, it means you were already in command mode.

If the screen is unreadable, press ⟨Ctrl⟩L (or ⟨Ctrl⟩R on some terminals) in command mode to make **vi** redraw the screen.

If you still cannot read the screen, either your terminal is set up incorrectly or you are editing a non-text file. Type **:q!** to exit without saving the current file. All the changes you made since the last save operation are discarded.

Saving files and quitting vi

When editing a file, you are actually making changes to a copy of it that **vi** has created. After you have made several changes to a file, you can write these to the original file to update it. Before quitting **vi**, you must write all the changes to the file to save your work.

To save a file and/or leave **vi** you must switch to command mode, if you are not already in it. You can always enter command mode by pressing ⟨Esc⟩ until the terminal beeps or flashes at you.

There are several ways to save files and leave **vi**, each of which begins with you typing a colon character (:):

:w   Save the current file (write file) but do not exit. This command fails if the file is read-only. You can save under a different name by adding a filename: for example, **:w** *newfile* saves the current file as *newfile* if that file does not already exist. The command **:w** writes to the file if it already exists but fails if it is read-only. Use **:w!** to overwrite a read-only file. (The exclamation mark tells **vi** to ignore any error conditions.)

:q   Quit **vi**. This command fails if you have made changes to a file since the last time you saved it. (If you really want to quit without saving, type **:q!**. This causes **vi** to quit without saving the current file.)

:wq   Save the current file and exit **vi**. The command **:x** is equivalent to this, except that it only saves the current file if you have changed it. These commands fail if the current file is read-only, or you are editing more than one file. See "Editing more than one file" (page 146) for details. (For information on read-only files, see "Access control for files and directories" (page 121).

Moving around a file

To move the cursor a single character in any direction, use the arrow keys. Alternatively, if you are in command mode, you can use the following keys:

h   move left one character

l   move right one character

k   move up one line

j   move down one line

You can also move around in larger units than a single character. To repeat any of the following movement commands (that take an optional parameter *n*) type the number of times you wish the command to occur followed by the command. For example, to move right five words, enter **5w**:

[*n*]**b**   move back one (or *n*) words

[*n*]**w**   move forward one (or *n*) words

^   move to the start of the line

$ move to the end of the line

[*n*]**(**   move back one (or *n*) sentences

[*n*]**)**   move forward one (or *n*) sentences

[*n*]**{**   move back one (or *n*) paragraphs

[*n*]**}**   move forward one (or *n*) paragraphs

(**vi** considers a sentence to be a sequence of characters ending with a dot, question mark, or exclamation mark, followed by either two spaces or a new-line. Sentences begin on the first nonwhitespace character following a preceding sentence, and are delimited by paragraph and section delimiters. A paragraph is any block of text delimited by empty lines or an **nroff** formatter macro.)

Other commands control the portion of the file that the screen displays:

⟨Ctrl⟩**U**   move back half a screen

⟨Ctrl⟩**D**   move forward half a screen

⟨Ctrl⟩**B**   move back one screen

⟨Ctrl⟩**F**   move forward one screen

You can specify line numbers to set your position within a file.

4

To discover the current line number, press ⟨Ctrl⟩**G**. A status line appears at the bottom of the screen, telling you the name of the file, whether it has been modified, your current line number, the number of lines in the file, and your position in the file as a percentage of the length of the file. For example:

```
"soliloquy" [Modified] line 24 of 35 --68%--
```

To make line numbers appear at the left-hand side of the screen, enter **:set number** (the numbers are not added to the text of your file). To make **vi** always display line numbers, you can add this command to your *.exrc* file. See "Configuring vi" (page 152) for more information.

The following commands move you to a specified line number:

#G   go to line number #

1G   go to the start of a file

G   go to the end of a file (equivalent to **$G**)

For details of the many other movement commands available see the **vi**(C) manual page.

Deleting and restoring text

To delete text, use the **d** command followed by the unit of text to delete. To delete several units of text at once, enter the number of items to delete (*n*) followed by the deletion command. The deletion commands available are:

[*n*]**x**   Delete letter under the cursor. **12x**, for example, deletes 12 letters.

[*n*]**X**   Delete letter to the left of the cursor.

[*n*]**dw**   Delete word from the cursor up to the next word including any white space. **4dw**, for example, deletes four words.

[*n*]**dd**   Delete the current line. **8dd**, for example, deletes eight lines.

[*n*]**dG**   Delete from cursor to the end of the file.

[*n*]**:**x,y **d**   Delete lines x through y of a file. For example, **:100,200 d** deletes lines 100 through 200, **:1,. d** deletes from the start of a file to the current line, and **:.,$ d** has the same effect as **dG**.

Restoring deleted text

You can always undo the last deletion by using the **u** or **U** (undo) command.

The command **p** (paste) inserts the last piece of text that you deleted to the right of the cursor. This is useful if you need to transpose two adjacent characters: position the cursor on the first character and enter **xp**. To swap two lines, place your cursor on the first, and enter **ddp**.

You can restore deleted text *before* the cursor by using the **P** command (upper-case) instead of **p** (lowercase). You can use this to swap two words that are on the same line; place the cursor on the first character of the first word, and enter **dwwP**.

Using the deletion buffers

vi remembers the last nine pieces of text that you deleted in *deletion buffers* numbered 1 for the most recent deletion, 2 for the next most recent, and so on. To restore the contents of a deletion buffer below the line on which the cursor is positioned, use the paste command with the *number* of the deletion buffer, *"numberp*. For example, typing **"2p** pastes the second most recently deleted piece of text at the cursor.

If you switch to editing another file (using **:n** or **:r**), or reload the original file (using **:rew!**), the contents of the deletion buffers are preserved so that you can cut and paste between files. The contents of all buffers are lost, however, when you quit **vi**.

There are also 26 named text buffers that you can use in **vi**, these are described in "Using buffers to cut and paste text" (page 146).

Searching for text

When searching for text, it is worth considering if you will want to return to your current position in the file. If so, you should either make a note of the current line number reported using ⟨Ctrl⟩G, or put a marker in the file using the **m** command. See "Placing markers" (page 147) for further details.

To start a search:

1. Type a slash (/); this takes the cursor to the bottom of the screen.

2. Enter the text you want to find and press ⟨Enter⟩. **vi** searches forward through the file looking for a matching text string. The cursor moves to the position of the next occurrence of text that matches.

3. To find the next match, press **n**. Repeat this until you find the match you want.

Press **N** to search backward through the file instead of forward.

Assuming that you are at the top of the example file (you could enter **1G** or **:1** to go to the first line), and you want to find all occurrences of the word "sleep". You enter **/sleep** to find the first match:

```
And by opposing end them. To die, to sleep -
```

4

Pressing **/** and ⟨Enter⟩ subsequently finds the following lines in turn:

```
No more - and by a sleep to say we end
Devoutly to be wished. To die, to sleep -
To sleep - perchance to dream: ay, there's the rub,
For in that sleep of death what dreams may come
```

This continues until the first occurrence in the file is found again:

```
And by opposing end them. To die, to sleep -
```

vi allows you to look for more general patterns of text using *regular expressions*. For a discussion of the regular expressions matched by **vi**, see "Editor regular expressions" (page 317) and the **regexp**(M) manual page.

Replacing and modifying text

To replace a single letter with another letter, position the cursor over the letter and type the **r** command, followed by the replacement letter. For example, to replace the "u" in "rub" by "i", move the cursor to the "u" and enter **ri**.

To replace an unlimited amount of text with new text, position the cursor over the first letter and type **R**. You are now in replace mode. This functions like insertion mode, but characters you type will replace the previously existing text until you return to command mode by pressing ⟨Esc⟩. For example, to convert "needle" into "noodle", move the cursor onto the first "e" and enter **Roo**⟨Esc⟩.

Use the ˜ command to switch letters between uppercase and lowercase text. First place the cursor over the character to be switched.

Join two lines by typing the command **J**. The line below the cursor is joined to the end of the current line. Any leading spaces on the lower line are replaced by a single one. For example:

```
Once more unto
        the breach
```

This produces the line:

```
Once more unto the breach
```

Substituting text

To substitute one sequence of characters for another on the current line, use the **:s/*old*/*new*/** command, where *old* is the sequence to find, and *new* is the sequence to replace it with. For example:

:s/needle/bodkin

This changes the first occurrence of "needle" when applied to the following line:

```
With a bare needle? Who would needles bear,
```

After the substitution it reads as follows:

```
With a bare bodkin? Who would needles bear,
```

Pressing **&** repeats the substitution on the current line:

```
With a bare bodkin? Who would bodkins bear,
```

Entering **:s/bodkins/fardels/** amends the line to give the correct quotation:

```
With a bare bodkin? Who would fardels bear,
```

If you wish to keep the old string as part of the new, you can refer to it as " **&** " within the new string. For example:

```
Tomorrow and tomorrow,
```

Applying **:s/tomorrow/& and &/** to the above line transforms it into the following:

```
Tomorrow and tomorrow and tomorrow,
```

Performing global substitutions

To substitute all occurrences of a sequence of characters within a file, type:

:g/*old*/s/*old*/*new*/g

The first element of this command, **:g/*old*/** is the address. **g/** (short for *global*) indicates that the following action will be applied globally (throughout the file) to all lines containing the string *old*.

The second element of the command, **s/*old*/*new*/g**, is the action. In this case, *new* will be substituted for *old*, globally, on the selected lines.

This command can also be written as follows:

:1,$s/*old*/*new*/g

In this form, the address is the range of lines **1,$**, where $ is an abbreviation for the last line in the file.

To restrict the change to a line or range of lines, replace the search command **/old** with the line number or range of line numbers to apply the command to. For example, to carry out the command on lines 5 to 20 of the file, type the following:

> *:5,20 s/old/new/g*

vi remembers the last string it searched for; the *empty* search command **//** matches all occurrences of the previous search string. For example, in the following command, the change is applied to lines containing *old*:

> *:g/old/s//new/g*

Because the target of the substitution is the empty search field, all occurrences of *old* are replaced (because *old* was the last item searched for).

You can search for regular expressions as defined in "Editor regular expressions" (page 317), but should replace them with a string of ordinary text. For example, to search for any single word beginning with "cent" (such as center, centered, or central) and replace it with "middle":

> **:g/cent[a-z].*/s//middle/g**

(Note that the ".*", representing a sequences of zero or more characters, is necessary to match the rest of the word. Otherwise the string matched by "cent[a-z]" may be replaced by "middle" but the suffix of the original word will not be removed by the substitution.)

Using the soliloquy example introduced earlier in this chapter, we can replace all instances of "Which" that start a line:

> **:g/^Which\>/s//That/**

The use of the "\>" notation ensures that the substitution is applied only to single words. The final **g** was left out as there can only be one beginning to a line. After performing this command, **vi** leaves you at the last line on which it performed a substitution:

```
That patient merit of th' unworthy takes,
```

You can apply a global substitution to all lines that do *not* match the search string, by starting the command with **:g!/** instead of **:g/**.

For example, suppose you have a file containing paragraphs of text separated by blank lines, and want to indent each line of text by one tab space. You do not want to add tabs to the blank lines. To do this, you can use the command:

> **:g!/^$/s/^/⟨Tab⟩/g**

(⟨Tab⟩ is a tab character). The regular expression ^$ matches an empty line; that is, a start-of-line metacharacter "^" followed immediately by an end-of-line metacharacter "$". The **:g!/^$/** command matches all lines that *don't* match this regular expression (that is, all nonempty lines). **vi** then executes the substitution command **s/^/⟨Tab⟩/g** which searches for the beginning of the line and inserts a tab character.

You might also wish to restrict the lines that are substituted. For example, applying the command **:1,$-1 s/^.*$/⟨Tab⟩&/g** to the file *soliloquy* indents every nonempty line except the last:

> .
> .
> .

```
        And enterprises of great pitch and moment,
        With this regard their currents turn awry,
        And lose the name of action. - Soft you now,
        The fair Ophelia! - Nymph, in thy orisons
        Be all my sins remembered.

    by William Shakespeare
```

When the command finishes, your position in the file is at the last line that was changed.

Specifying addresses

The following types of address are recognized:

.	The current line. (This is the default address for most **vi** commands.)
57	The fifty-seventh line in the current file.
57,4711	Lines 57 through 4711.
$	The last line in the file.
1,$	Every line in the file.
±5	The fifth line after the current one. (Negative offsets are also permitted; for example, -8 refers to the eighth line before the current one.)
.,+8	The current line to the eighth line after it. (Negative offsets are not permitted in this case.)

/pattern/ Matches the first line after the current that contains the regular expression *pattern*. See "Editor regular expressions" (page 317) for details of the regular expressions **vi** recognizes.

To match any single word beginning with some prefix *prefix*, you should search for the regular expression \<*prefix*[a-z]*, which matches *prefix* followed by any sequence of lowercase letters. If you search for \<*prefix*.* it will match from *prefix* to the end of the line. This may cause unpredictable results when substituting text.

`*marker* Matches the line containing the *marker* placed using the **m** command. See "Placing markers" (page 147) for more details.

Confirming substitutions

To make **vi** prompt you for confirmation before it changes each instance of a string, use the **gc** command. You might want to do this if you only want to replace some occurrences of a word in a file, rather than all of them. For example, to substitute "hawk" for "handsaw" throughout a text, prompting for confirmation before each change, enter:

>   :g/hawk/s//handsaw/gc

The command is broken down as follows:

g/hawk/   Applies the change to every line containing the string "hawk".

s//   Makes a substitution; the empty search target // means that **vi** will carry out substitutions on the previous search string ("hawk").

handsaw/gc   Replaces the search string with "handsaw" everywhere (**g**), subject to confirmation of each change (**c**).

In response to this command, **vi** prompts you at each occurrence of the word "hawk", printing the line it occurs on. Enter "y" to make the substitution; enter "n" to ignore the occurrence and continue the search. (If you enter something which **vi** does not recognize, it plays it safe and does not carry out the substitution.)

Repeating and undoing commands

To repeat the last insertion or deletion command, type a dot (.); the last action will then repeat. For example, to delete three lines, go to command mode and type:

dd..

The **dd** command deletes a line, and each period repeats the command.

You can tell **vi** to carry out a command a number of times by first entering the number, then the command. For example, to delete five lines starting with the line the cursor is on, enter the command **5dd**. Type **u** to undo this command. Type **u** again to undo the undo.

Including the contents of another file

To read a file into **vi** at the current cursor position, switch to command mode and press **:r**, then type the name of the file. If the file is readable, its contents will be inserted below the cursor.

You can use addresses with this command, including 0 to specify a dummy line before the first line in the file. For example, if you want to add the file *preamble* at the top of *soliloquy*, used in a previous example, you would enter **:0r preamble** to produce:

```
Hamlet's soliloquy from Act III Scene 1:

        To be, or not to be: that is the question:
        Whether 'tis nobler in the mind to suffer
        The slings and arrows of outrageous fortune,
    .
    .
    .
```

The command **:r** loads another copy of the last-saved version of the present file below the current line.

Accessing the shell

To include the output from a system command, you would enter the following:

:r !*command*

This inserts the output below the current line. Alternatively, to replace the current line, type the following:

!!*command*

For instance, **:r !date** includes the date and time below the current line; **:$r !date** puts the date and time at the end of the file. See also "Running other programs from inside vi" (page 148).

Editing more than one file

To edit more than one file in a session, start **vi** giving it a list of files, as follows:

> **vi** *file1 file2 file3 ...*

vi only edits one file at a time, but you can move forward through the list by typing **:n** (next) in command mode, or go to a specific file by typing **:e** *filename* (short for **:edit** *filename*) in command mode. Note that you cannot edit the next file if you have made changes to the current one, unless you save the current file with **:w** or override **vi** by using the **:n!** (unconditional next file) command. To return to editing the first in the list of files, type **:rew** (short for **:rewind**).

Note that if you switch files without saving the contents of the last one you edited you will lose any changes you have made: see "Saving files and quitting vi" (page 136) for how to save the current file.

Using buffers to cut and paste text

vi has 26 buffers, named " a " through " z ", where it can store text temporarily. You can use buffers to:

* store text for insertion elsewhere in the current file

* store text to be inserted in a newly loaded file

* store frequently typed phrases to speed typing

The contents of buffers disappear altogether when you quit **vi**.

To cut (delete) a line of text and store it in a buffer, type **"***buffer***dd**.

To copy (yank) a line of text into a buffer, type the command **"***buffer***yy** or **"***buffer***Y**. For example, **"iyy** copies the current line into buffer " i ".

To store several lines, precede the cut or copy command with the number of lines you wish to copy; for example, to copy fifteen lines into buffer " j ", type **"j15Y**.

To paste the contents of the last buffer you used into the text on the line immediately below the cursor, type the command **p**. You can insert the contents of any buffer by specifying the buffer name: **"***buffer***p**. For example, to paste the contents of buffer " g " into your file below the cursor, type **"gp**. The command **"gP** places the text above the cursor.

Placing markers

A marker behaves like a bookmark; it saves your place in a file so that you can return to it from anywhere in the file. **vi** allows you to use up to 26 markers, named " a " through " z ". To place a marker at the current cursor position, type **m** (for mark) followed by the letter that identifies it. To go to the place in a file where you have set a marker, enter a single back quote (`) followed by the marker's letter.

For example, you could type **mh** occasionally to keep your place in a very long file. If you then search for a piece of text elsewhere in the file, you might not be able to remember where you started the search. Enter `h and you return there immediately.

If you reload the file (using **:rew** or **:rew!**), or you load a different file, the markers are lost. Similarly, you lose a marker if you delete the line it was set on, and restore the line at a different place in the file.

Using keyboard shortcuts

vi provides the following powerful mechanisms for speeding up your work:

Running other commands
Suspend **vi** temporarily while you run another command, such as a shell. This avoids having to save your file, quit **vi**, run the command, and then restart **vi**.

Using a filter
Send a section of your document through a filter program, and have the output from the filter inserted into your document in place of the original text.

Abbreviations
Define a short abbreviation for a long series of characters in text-entry mode.

Named buffer execution
Make **vi** execute the commands stored in a named buffer as though you had entered them.

Keystroke remapping
Link a sequence of commands to a key; when this is pressed, **vi** carries out the commands.

Running other programs from inside vi

The command :!*cmd* executes the program *cmd*, then returns to **vi**. For example, :!**sh** starts a new shell without exiting **vi**; when you quit the shell (by typing **exit** or ⟨Ctrl⟩**D**), you return to **vi**.

Sending text through a filter

You can send some or all of the contents of the current file through a program that acts as a filter, transforming the contents of the current file. For example, to use the **tr** command to translate the current paragraph into uppercase:

1. Enter command mode by pressing ⟨Esc⟩, if necessary.

2. Go to the beginning of the first line of the paragraph.

3. Type the following characters:

    !} **tr** '[a-z]' '[A-Z]'

The command ! tells **vi** to filter all the text from the current cursor position to the position indicated by the subsequent movement command (in this case " } ", the command to move to the end of the paragraph), through the program **tr** with the arguments '[a-z]' '[A-Z]'. **tr** translates its input, defined by the first wildcard expression, into an output file defined by the second expression. The output from **tr** is then substituted for the input.

In general, you can use any filter in this way:

    ! *movement filter* ⟨Enter⟩

An exclamation point typed in command mode introduces a filter command. *movement* is a **vi** command to move the cursor to the end of the block of text you want to feed through *filter*. The cursor jumps to the bottom line of the screen when you enter the movement command, and a " ! " prompt appears; **vi** waits for you to enter the filter command and press ⟨Enter⟩. Note that the cursor movement must refer to a block of text larger than one word; the commands **w** and **e** are not acceptable cursor movements.

Another common usage is to spell-check a sentence using **spell**, and replace it with a list of all the unidentified words it contains:

1. Enter command mode by pressing ⟨Esc⟩, if necessary.

2. Move to the start of the sentence by pressing 0.

3. Enter the command:

    !)**spell** ⟨Enter⟩

The **)** command moves the cursor to the end of the current sentence. The **!)** command selects the text from the current cursor position to the end of the sentence and sends it to the standard input of **spell**. The output from **spell** (all the words in the sentence that it cannot find in its dictionary) replaces the sentence and those below it with each mispelled word that is found. Retrieve the original sentence using the **u** (for Undo) command immediately. For example:

```
When looking for occurences of spelling mistakes
```

Applying the above procedure replaces this sentence with the following:

```
occurences
```

The escape filter can be used with any command that reads from the standard input. For example, use **wc** to find the number of words in a file; go to the top of the document and issue the command **!Gwc -w**. The document is replaced by a word count. Press **u** to get your document back again.

Defining abbreviations

To define a short abbreviation that, when typed, is replaced by a longer word or phrase, use the **ab** command. For example, to define **eC** as an abbreviation for **European Community**, enter the command:

> **:ab eC European Community**

From now on, whenever you type the letters "eC" while inserting text, **vi** will expand them into the phrase "European Community". **vi** waits until you finish inserting text or type ⟨Space⟩ or ⟨Tab⟩ before making the expansion.

Note that the name of the abbreviation cannot contain any space.

It is a bad idea to use a single letter as an abbreviation for a word; every time that letter is typed, it will be replaced. It is also a bad idea to use common two-letter combinations like "ch" or "ee" or "th".

You can prevent an abbreviation from being expanded by escaping the first character following it. For example, the word "eC" will not be replaced by "European Community" when you type it, if you follow it immediately with a ⟨Ctrl⟩**V** (which escapes the next character). (You can insert any control character by pressing ⟨Ctrl⟩**V** followed by the control character itself. You can also embed a ⟨Enter⟩ character inside a command line in this way.)

To remove an abbreviation, use the **unab** command. For example, to clear the "eC" abbreviation, go to command mode and type:

> **:unab eC**

If you subsequently type the letters "eC" they will not be expanded. To examine the currently defined abbreviations, type **:ab** with no arguments.

4

Storing a command in a buffer

You can store a frequently executed command sequence in a named buffer, and execute it with the command *@buffer* (*buffer* is the name of the buffer, such as " t "). For example, you might want to create a sequence to place you at the end of the file you are editing, and enable you to return to your current location. In this case, carry out the following steps:

1. Open a new line using **o** and enter the following sequence of characters:

    maG

    These correspond to the commands to place marker " a " at the current position in the file and to go to the end of the file.

2. Leave text insertion mode by pressing ⟨Esc⟩.

3. Copy the line into a buffer using the following:

    ^"tdd

    The ^ moves the cursor to the start of the line, and **"tdd** deletes the line and stores it in buffer **t**.

When you issue the command **@t**, **vi** reads the contents of buffer " t " and treats it as a command typed at the keyboard; **vi** places marker " a " in the text, then goes to the end of the file. (All you need to do to return to your current location is to type the command `a, which is too short to be worth assigning to a key.)

Mapping key sequences

vi can assign sequences of commands to a series of keys or control characters so that whenever a mapped key sequence is typed, the command to which it is mapped is carried out. There are two commands to do this: **:map**, which works in command mode, and **:map!**, which works in insertion mode.

Assigning commands to a key sequence

To assign a command to a sequence of keystrokes, issue the **:map!** command in command mode. For example, to assign the command **"ad$** to the key ⟨Ctrl⟩C, you would enter the following:

>   :map ^C "ad$

^C is a ⟨Ctrl⟩C character. (To enter the ⟨Ctrl⟩C without letting **vi** interpret it, type ⟨Ctrl⟩V⟨Ctrl⟩C.) This command cuts all text between the current cursor position and the end of the line, and places it in buffer "a". Now, whenever you type ⟨Ctrl⟩C in command mode, it will carry out the defined series of commands.

> **NOTE** It is a bad idea to assign mappings to key sequences that are already in use as commands, because it renders the commands inaccessible.

Assigning insertions to a key sequence

To assign a key mapping in insertion mode, go to command mode and issue the **:map!** command. For example, to make the sequence "EC" map to "European Community", enter:

> **:map! EC European Community**

To remove a mapping from a key, use the **unmap** (or **:unmap!**) command; for example:

> **:unmap ^C**
> **:unmap! EC**

When you begin to type a letter that may be part of a mapped command, **vi** saves up your typed input and checks whether it matches a mapping before it does anything. If the key sequence you type is a mapped command, **vi** executes it as soon as the command is complete; if not, when it becomes clear that you are not typing a mapped key sequence, **vi** acts on the keystrokes. (**vi** waits for a period of time that you can alter by resetting the **timeout** internal variable. See the **vi**(C) manual page for further details. The **ab** command works in a different manner described by "Defining abbreviations" (page 149).)

To stop **vi** from expanding a mapping, you must escape the first character of the command *before* you begin typing. For example, using the mapping above, to enter the acronym "EC" you must escape the mapped word by entering:

> **⟨Ctrl⟩VEC**

(If you had used an *abbreviation* for "ma1G" instead of a mapping, you would have to type the following to escape it:

> **ma1G⟨Ctrl⟩V**

> **NOTE** Never define a mapping that contains its own name.

For example, consider the following bad definition:

> **:map! n banana**

If you were to try this example, the next time you typed the letter "n" **vi** would expand it to "banana". But it would only get as far as the third character in "banana" before it saw another "n" and tried to expand it. The process produces output like:

```
babababababababababababababababababababa
```

vi begins to execute the mapping then dies.

Changing modes within a mapped command

Sometimes it is useful to create a mapping or command that changes mode while it runs. For example, to create a command-mode mapping that inserts a piece of text under the cursor without leaving command mode, do this:

    :map h iHello there!^[

The " ^[" character means "escape". To enter it into a mapping without letting **vi** act on it as if it is a command, press ⟨Ctrl⟩**V**⟨Esc⟩.

When you press "h" in command mode, **vi** carries out the actions in the mapping. It executes the command **i** and switches to insert mode, enters the characters "Hello there!", then switches back to command mode on encountering the Esc character in the sequence of keys.

For example, suppose you have stored your name and address in a file called *.address* in your home directory. You can define a mapping that will automatically insert the contents of *.address* at the beginning of a file you are editing. For example, if your login is *roberta*, the mapping would be:

    :map % ma1G^:0r /u/roberta/.address^M`a

When you type the "%" key in command mode, the first command, **ma**, marks your current position in the file with marker "a". The next commands, 1G^, go to line one of the file, then to the beginning of that line. The following command *read*s in the file */u/roberta/.address* at the current location:

    :r /u/roberta/.address^M

The mapping ends with the command `a, which returns to marker "a".

Note that ^M is a carriage return character entered into your mapping by pressing ⟨Ctrl⟩**V**⟨Enter⟩.

Now whenever you type the "%" key in command mode, **vi** executes the command sequence to insert */u/roberta/.address* at the top of your file.

Configuring vi

vi has a number of internal variables. These can be configured by typing the **:set***varname* command, where *varname* is the name of the variable to change.

To examine **vi**'s current settings, go to command mode and type **:set all**.

If a variable name starts with "no", it is not set (that is, not switched on). You can set it by typing **:set** *varname*, with an optional value. If a variable name does not start with "no", and is not followed by a number, it is set. You can switch it off by issuing a command like **:set no***varname*. For example, to make **vi** ignore wildcards, you must switch off the variable "magic". To do this, type **:set nomagic**.

To make **vi** automatically begin a new line before you reach the right-hand side of the screen, type **:set wrapmargin=15**. The first word that is less than fifteen characters from the right-hand side of the screen is placed on a new line. (If you have used word processors, this feature may be known to you as "word wrap".)

Here are some of the most useful **vi** settings. Some of them can be abbreviated; for example, typing **:set autowrite** and **:set aw** have the same effect.

number   Displays line numbers at the left-hand edge of the screen. (The numbers are not part of the saved file). You can go to any line by going to command mode and entering the line number followed by **G**. (This setting can be abbreviated to **nu**.)

autoindent

Indents the left-hand margin of new lines of text by an amount determined from the previous line of text. For example, if you indent a line by one tab, **vi** will automatically indent all subsequent lines by the same distance until you cancel the previous indent by pressing ⟨Ctrl⟩**D**. (This setting can be abbreviated to **ai**.)

autowrite Saves any changes that have been made to the current file when you issue a **:n, :rew**, or **:!** command. (This setting can be abbreviated to **aw**.)

ignorecase

Ignores the case of text while searching. (This setting can be abbreviated to **ic**.)

list   Prints end-of-line characters as "**$**", and tab characters as "**^I**". These characters are normally invisible.

showmode

Displays a message at the bottom right of the screen when you switch to a text input mode.

tabstop   Sets the number of spaces between each tab stop on the screen. When you press the ⟨Tab⟩ key in insertion mode, you are inserting a special *tab character* into your text. **vi** interprets this as a command to replace the tab character with however many spaces are needed to bring the cursor into line with the next tab stop position. **vi** normally puts tab stops eight characters apart; by using **:set tabstop=***number* you can change the number of characters between tabs. This is useful for adjusting the width of columns of text or levels of indentation in documents created using tabs instead of spaces. (This setting can be abbreviated to **ts**.)

4

A complete list of the internal **vi** variables and their meanings is included in **vi**(C).

You can create a list of these variables that are automatically set whenever you start **vi** as described in the following section.

Saving frequently used commands

If you have a set of commands that you use frequently, place them in a file called *.exrc* in your home directory. Whenever **vi** starts up it will look for this file and execute any commands or mappings it sees in it. The mappings will be available to you in every **vi** session without the need to retype them each time.

You can place comments in your *.exrc* file to tell you what your commands do, by starting a line with a " # "; this tells **vi** to ignore the line. For example, your *.exrc* file might look like this:

```
# My .exrc file. Last changed: 20-March-1995
#
# Define where new lines are to be inserted automatically
:set wrapmargin=15
# Allow special characters in search patterns without using backslash
:set nomagic
# Define an abbreviation
:ab eC European Community
# The % key in command mode now reads in the file .address
# at the top of the current file
:map % ma1G^:r /u/roberta/address^M`a
```

Using ed

ed is a line editor; it edits a single line at a time. Occasions on which you might need to use **ed** include:

- working on a incorrectly set terminal

- no other text editor is available

- executing a script of editing commands automatically

- processing a script of editing commands output by the file comparison command, **diff**(C)

Starting ed

From the shell prompt, type **ed** *filename*, where *filename* is the file to edit. **ed** starts, then prints the number of characters it has read from *filename*. You are in command mode.

Saving files and quitting ed

To save a file, use the commands:

w   saves the current file

w *filename*
       saves the current file as *filename*

To quit **ed**, use the commands:

q   quits **ed** only if the current file has been saved

Q   quits **ed** without checking if the current file has been saved

See the **ed**(C) manual page for more details of these commands.

Moving around in ed

Instead of showing you a screen of text, **ed** works with *line addresses*; the line or lines in a file to which it applies a command.

When you start **ed**, it displays the number of bytes in the file. You begin at line 1; to move to a different line, enter its line number and press ⟨Enter⟩. **ed** echoes the contents of the line. Press ⟨Enter⟩ to step through the file a line at a time. You can also enter relative line numbers; for example, -2 to go back two lines, +5 to go forward five.

The special address " . " refers to the current line, " $ " to the last line of the file.

To discover the current line number, enter the command **n**. This also outputs the contents of the line.

To see the contents of your file, use the l (short for **list**) command. Entering l on its own prints the current line. To list several lines, prefix l with the start and end line numbers separated by a comma (,), For example, to list lines 10 to 20 of a file, enter:

       10,20l

This command sets your current line number to the last line displayed (20).

For convenience, a comma " , " represents the address pair "1,$" (that is, the entire file), while a semicolon (;) stands for ".,$" (current line to end of file). For example, ,l lists the entire file.

You can use *relative* addresses; for example, **$-5** means the fifth line before the last line of the file, while **.+2** means the second line after the current line.

Editing text in ed

There are several commands for editing text in **ed**:

a   Appends text. If the file is new, you can only enter text using this command. If the file already exists, specify an address (in the form of a line number or search pattern) to select the line to which to append text.

c   Changes text. This command requires the address(es) of the line(s) to be replaced. When you finish entering new text, press ⟨Ctrl⟩**D** or " ." on a new line to return to command mode.

d   Deletes text. This command requires the address(es) of the line(s) to be deleted.

i   Inserts text. Whatever you type is inserted before the current line; press ⟨Ctrl⟩**D** or " ." on a new line to stop inserting text and return to command mode.

s/*old***/***new*

Replaces text. **ed** replaces the first occurrence of *old* on the line with *new*. The string *old* can contain regular expressions, as with **vi**.

If you prefix this command with a range of addresses, each line in the range is searched and the first instance of *old* on each line is replaced with *new*. For example, **,s/foo/bar/g** replaces every instance of "foo" on every line by the string "bar".

u   Cancels the effect of the previous command.

ed supports the same regular expressions as **vi**; see "Editor regular expressions" (page 317). For full details, see also **regexp**(M).

Chapter 5
Controlling processes

The UNIX system is designed to hold many programs in memory. Although the computer can only execute one program at a time, by swapping between programs frequently it can maintain the illusion that it is running them simultaneously.

A program that is being executed by the UNIX system gives rise to a *process*. This chapter contains the following information about processes:

- what is a process? (this page)
- finding out what processes are running (page 158)
- background jobs and job numbers (page 160)
- killing a process (page 162)
- keeping a process running after you log off (page 165)
- using signals under the UNIX system (page 166)
- reducing the priority of a process (page 167)
- scheduling your processes (page 169)

What is a process?

Processes are not the same as programs; in addition to the machine instructions (often called "text"), they have additional components (mainly data that is being processed in memory) that are not part of the program itself.

Several processes being scheduled for "simultaneous" execution by the kernel may in fact be instances of a single program. For example, on a multiuser system, several users may use the **cat** utility: this is held as a single program in */bin/cat*.

For an explanation of how the system manages processes, see Appendix A, "An overview of the system" (page 393). See also "Understanding the UNIX system" (page 429).

You may need to destroy runaway processes, or processes that have finished running but have not been removed from the system ("zombies"). You may also need to find out what processes are running, cause processes to run after you log off, and execute groups of processes (piping the output of one to the input of another). These tasks are explained below.

There are two methods for managing processes; process control and job control. Process control allows you to interact with all the processes on the system. Job control allows you to move jobs between the background and foreground using the shell.

Finding out what processes are running

To find out what processes are running, use the **ps** command (process status) which prints information about the processes associated with your terminal (that is, the processes from your current login session).

To find out all the processes running on the system, type the following:

```
$ ps -ef
    UID   PID  PPID  C   STIME TTY      TIME COMMAND
   root     0     0  0  Sep 24 ?        0:00 sched
   root     1     0  0  Sep 24 ?      110:56 /etc/init
   root     2     0  0  Sep 24 ?        0:00 vhand
   root     3     0  0  Sep 24 ?        5:52 bdflush
  gavin  8501     1  0 17:59:05 004     0:03 -ksh
  gavin  8972  8501  0 18:52:04 004     0:02 vi tmpfile
   root   423     1  0  Sep 24 02       0:00 /etc/getty tty02 m
susanna  7903     1  0 17:29:01 015     0:04 -csh
  perry  8608     1  0 18:12:27 006     0:06 -ksh
```

Note that there may be other processes running on the system that you are not authorized to see. (You will probably have to pipe the output of **ps -ef** through **more**(C) or **pg**(C), as several hundred processes may be reported on a large system.)

The listing contains the following columns:

UID   the user name of the owner of the process

PID   the process ID

PPID   the parent process ID

C   scheduling information (of interest to administrators investigating performance problems)

STIME   the time when the process was started

TTY   the terminal to which a process is attached; for example, user
            perry is working on tty006

TIME   the cumulative time for which the process has been executed

COMMAND   the command that resulted in the creation of the process

The **ps** command supports many more output columns, controlled by the
command line options; for details, see **ps**(C).

As soon as it is created, each process is allocated a unique identifier called a
process ID or PID, a decimal integer in the range 0-65535. Some of these are
reserved for the system. On system startup, a process called *sched* is created
by the kernel; this creates three other processes called */etc/init*, *vhand* and
bdflush. These four processes are automatically allocated process ID's 0, 1, 2
and 3 respectively. It is *sched*, the "swapper" process, that swaps other pro-
cesses into main memory before the kernel scheduler can allocate CPU time to
them.

Under the UNIX system, all processes (except *sched*) are created by a pro-
cedure known as "forking". The process that does this is known as the
"parent" of the resulting "child" process. The relationship between a parent
and a child can be identified by the process' parent process ID (PPID). Each
process (except *sched*) has a single parent process, but may have many child
processes. In the example, the **vi** process (8972) was created by process 8501,
which was in turn created by */etc/init*. **init**(M) is the ancestor of all other pro-
cesses active on a UNIX system: among other things, it calls a program called
getty(M), which is responsible for creating login processes, which in turn calls
up a user shell such as **ksh**(C) or **sh**(C).

Process creation is known as forking because the calling process splits in two.
The copy is created by calling the **fork** function. The child is an almost exact
copy of its parent, made by assigning a slot in the process table to the new
process, then copying information from the parent's process table slot to the
child's slot. The obvious differences between the parent and its child are the
PID and PPID. See **fork**(S) for more details. A successful process creation is sig-
naled by the **fork** function passing a value of zero to the child process and the
PID of the child to the parent.

PID allocation in the 0-65535 range is cyclical: once the upper limit has been
reached, the lower numbers are reused, subject to the proviso that PIDs must
be unique.

To see what processes a particular user is running, type **ps -u** *login* where *login* is the login of the user in question. For example:

```
$ ps -u charles
  PID   TTY      TIME COMMAND
10170  008      0:07 ksh
 9779  008      0:00 ksh
 9780  008      9:23 pmview.r
 9791  p0       0:12 oadaemon
 9796  p1       7:47 email
 9797  p2       0:03 ksh
 9802  p6       0:02 ksh
19027  p5       0:02 ksh
19980  p6       0:20 vi
21275  p6       0:00 ps
$
```

Note that this user is running several Korn shell processes, each with a unique PID, but derived from the single program */bin/ksh*.

Background jobs and job numbers

You can run a process in the background, so that while it executes you can get on with something else, by appending an ampersand (&) to the command line, as follows:

```
$ vi tobermory &
[1]     13560
```

In this case, "1" is the job number and "13560" the process ID. Unlike PIDs, job numbers are allocated by the currently running shell, not by the operating system. While PIDs are assigned to all currently active processes, job numbers represent only active background processes, and while PIDs are unique across the system, job numbers are not.

Compound commands (where processes are linked using the semi-colon) should be grouped using parentheses, as follows:

```
$ (sleep 20; date > /dev/tty06) &
[1]     25143
$ jobs
[1] +  Running                (sleep 20; date > /dev/tty06)&
```

Waiting for background jobs to finish before proceeding

The **wait**(C) command is useful when you want to wait for the completion of background processes before performing a foreground command on them. For example:

```
$ spell file1 > spell_file1 &
[1]     13655
$ spell file2 > spell_file2 &
[2]     13657
$ wait
$ sort spell_file1 spell_file2 > spell_list
```

This script file runs the two **spell**(C) commands simultaneously in the background, waits for them both to complete, and then sorts the resultant files (*spell_file1* and *spell_file2*), putting the output in *spell_list*.

See also Chapter 11, "Automating frequent tasks" (page 245) for details of script files.

Finding out what jobs are running

To find out what jobs are running under your current shell, use the **jobs**(C) command, as follows:

```
$ jobs
[2] + Running            tar tv3 * &
[1] - Running            find / -name README -print > logfile &
$
```

A "+" in the second column signifies that the job has higher priority than a "-". The third column contains the state of the job; whether it is running, paused, waiting for input, or stopped. Finally, we see the command line that created the current job.

Use the **-l** option to **jobs** to display the PID after the job number. To limit the display to PID only, use the **-p** option.

```
$ jobs -l
[2] + 5236    Running        xrep *.s >bar &
[1] - 5195    Running        checkmac *.s >foo&
$ jobs -p
5236
5195
$
```

Killing a process

It is sometimes necessary to kill an executing process. This may be because it is taking up too much process time, causing the system to slow down, or perhaps because it is caught in a loop, and will therefore never complete. To kill the current process, try pressing:

- ⟨Ctrl⟩⟨Del⟩

- ⟨BREAK⟩

- ⟨Ctrl⟩**D**

- ⟨Del⟩

One of these actions should send an interrupt signal to the process (see "Using signals under the UNIX system" (page 166) for more information on signals).

Only the root user can kill processes belonging to another user or to the system.

If the process you want to stop is a child of your shell, you can use the **kill**(C) command and the process' job number to stop it, as in **kill %***jobnumber*.

If neither the interrupt generation keyboard sequences nor the **kill** command stop the process, try the following:

1. Log in to another terminal.

2. Use **ps -u** *your_login* to find out the process ID of the process you want to stop.

3. Type **kill** *pid* to kill the process.

For example:

```
$ ps -u charles
PID     TTY     TIME    COMMAND
8367    003     0:03    sh
6800    005     0:03    sh
26013   005     0:00    find
26073   003     0:00    ps
$ kill 26013
$ ps -u charles
PID     TTY     TIME    COMMAND
8367    003     0:03    sh
6800    005     0:03    sh
26073   003     0:00    ps
$
```

The **kill** command sends a signal to the target process (26013 in the above example) that causes it to halt. If you run **kill** without attaching a signal value to it, the signal is given a default value of 15, which is a command to "terminate," which will normally stop a process. If it does not, type **kill -9** *pid*. This is more effective, but does not give the process a chance to close any files it may be working on when it receives the signal.

In theory, if you stop a parent process, you automatically stop all the child processes spawned by it, as follows:

```
$ ps -ef | grep charles
charles 11487     1  0 08:34:15  008      0:07 -ksh
charles 11514     1  0 08:34:44  009      0:00 -ksh
charles 11641 11514  1 08:37:41  009      0:05 rm -r *
charles 11650 11487 11 08:38:32  008      0:00 grep charles
charles 11651 11487 71 08:38:32  008      0:01 ps -ef
$ kill -9 11514
$ ps -ef | grep charles
charles 11487     1  0 08:34:15  008      0:07 -ksh
charles 11650 11487 11 08:38:32  008      0:00 grep charles
charles 11651 11487 71 08:38:32  008      0:01 ps -ef
$
```

Note, however, that this procedure runs the risk of corrupting data or causing some other unwanted effect. As a general rule, always explicitly kill a child process before its parent:

```
$ ps -ef | grep charles
charles 11487     1  0 08:34:15  008      0:07 -ksh
charles 11514     1  0 08:34:44  009      0:00 -ksh
charles 11641 11514  1 08:37:41  009      0:05 rm -r *
charles 11650 11487 11 08:38:32  008      0:00 grep charles
charles 11651 11487 71 08:38:32  008      0:01 ps -ef
$ kill 11641
$ kill 11514
$ ps -ef | grep charles
charles 11487     1  0 08:34:15  008      0:07 -ksh
charles 11650 11487 11 08:38:32  008      0:00 grep charles
charles 11651 11487 71 08:38:32  008      0:01 ps -ef
```

It is also advisable to try killing a process with the lowest severity signal possible. Only use **kill -9** as a last resort.

The status of a killed process is reported as follows:

```
[1] + Killed    rm -r *
```

It is possible to kill all of the currently executing background jobs together, using the **kill -p** option and some of the special shell parameter notation described in "Passing arguments to a shell script" (page 250), as follows:

```
$ kill "$@" $(jobs -p)
```

Suspending a job

You can only suspend a job that was started in your current shell. To suspend a running job, press ⟨Ctrl⟩Z (or the current defined suspend key). The job will stop running, but will remain available (and can be continued).

For example:

```
$ sort bigfile.dat >bigfile.out
⟨Ctrl⟩Z
[1] + Stopped              sort bigfile.dat>bigfile.out
```

If ⟨Ctrl⟩Z does not work, your suspend key may not be set up correctly. To identify the suspend key, use the following command:

```
$ stty -a | grep susp
swtch = ^@;susp = ^Z;start = ^Q;stop = ^S;
```

This command line looks for the word susp in the **stty** settings. If it is present, it will tell you the current suspend key. If it is not present, or if you want to change it to another key, you can modify it using the **stty** command. For example, to make ⟨Ctrl⟩Q the suspend key, enter the following:

```
$ stty susp ^Q
```

(The "^Q" is entered by typing a caret (^) followed by a letter " Q ".)

Moving background jobs to the foreground

In order to terminate a background job using a keyboard interrupt sequence, you must first move it into the foreground. To do this, use the **fg**(C) command followed by a "%" sign, then the PID, the command name, or the job number. For example, to move the **find** background process to the foreground, type the following:

```
$ jobs
[2] +  Running              tar tv3 * &
[1] -  Running              find / -name README -print > logfile &
$ fg %find
find / -name README -print > logfile
```

Note that the command line is returned, without the trailing ampersand. The following command line has the same effect:

```
$ fg %2
find / -name README -print > logfile
```

The process now executes in the foreground; that is, it takes control of the terminal, and will accept input typed at the keyboard, including interrupt sequences such as ⟨Ctrl⟩Z.

fg entered without an argument, where only one background job exists, moves that job into the foreground. Where there are two or more background jobs, entering **fg** without an argument moves the job placed in the background most recently into the foreground.

Moving foreground jobs to the background

As we saw in "Killing a process" (page 162), to move a foreground process to the background, press the suspend key. The process is suspended and a message is displayed, as follows:

```
[1]  +  Stopped        sleep 30
```

To move a suspended job to the background, use the **bg**(C) command followed by a "%" symbol, then the PID, the job number, or the command name. For example, to move the suspended **sleep** process in the above example to the background, type the following:

```
$ bg %sleep
```

The suspended process is then restarted in the background. A message is displayed indicating this, as follows:

```
[1]    sleep 30&
```

Keeping a process running after you log off

It is sometimes useful to let one or more of your processes continue after you log out. In the Bourne and Korn shells, you can use the **nohup**(C) (no hangup) command to do this. (**nohup** is unnecessary in the C shell; background jobs started by **csh** continue even after the parent shell terminates.)

The format of **nohup** is as follows:

nohup *command*

Executed in the background using the ampersand, such a process will continue executing for its normal term, and will not be aborted by finishing the session.

For example:

```
$ nohup find / -name chapter6.txt -print > files_found &
```

The **find** process runs in the background and does not stop when you log off. The output is directed to a file called *files_found*. If redirection is not specified, output from the program is saved in a file called *nohup.out* in the current working directory, or in **$HOME**/*nohup.out* if the current directory is unwritable.

Using signals under the UNIX system

Signals are sent to processes by the UNIX system in response to certain events. Most signals cause the process receiving them to terminate abruptly. However, if you have set a "trap" for the signal, you can use them to recover from the emergency. You can also use signals as a means of allowing shell scripts to communicate with other programs running on the system. Note that signal 1 is not a real signal: it is sent by the shell and trapped within the shell for its own purposes.

The following signals are recognized by the shells (although not all of them can be trapped):

Table 5-1 Shell signal handling

Value	Signal	Description
0	EXIT	Exit from the shell.
1	HUP	A pseudosignal used by the shell, indicating that the standard output has hung up; sending this signal logs you out.
2	INT	Sent by a ⟨Del⟩ keystroke; sends an interrupt to the current program.
3	QUIT	Sent by a ⟨Ctrl⟩\ keystroke; causes the current program to abort, leaving behind a core dump (for use in program debugging).
9	KILL	Cannot be trapped or ignored; forces the receiving program to die.
11	SEGV	Indicates that a segmentation violation (a memory fault within the UNIX system) has occurred. This signal cannot be trapped or ignored by a shell; it invariably terminates the receiving process and causes a core dump.

(Continued on next page)

Table 5-1   Shell signal handling
(Continued)

Value	Signal	Description
15	TERM	Terminates the receiving program. (This signal should be used in preference to Signal 9, because the receiving program can catch it and carry out some shutdown operation, for example closing open files; Signal 9 forces the process to die immediately.)

For a full list of the UNIX signals, see **signal**(S).

You can use the **trap**(M) command to catch any or all of the trappable signals. Its format is as follows:

> trap *command signals_list*

In other words, *command* is executed whenever one of the listed signals (which are specified using the numbers given in the first column of Table 5-1, "Shell signal handling" (page 166)) is received. For example, if you have a shell script that uses a temporary file called *scratchpad*, the file will be left behind whenever the script is interrupted, unless you add the following line somewhere near the top of the script:

```
$ trap "rm scratchpad" 0 1 2 3 15
```

This statement deletes the temporary file whenever an EXIT, HUP, INT, QUIT or TERM signal is received.

Reducing the priority of a process

One of the factors that controls when a process is executed is its "priority", which can be manipulated using the **nice**(C) command. This is useful at the start of a big job such as compressing all the files in a directory, particularly when the speed at which the job completes is not crucial. **nice** has the following form:

> nice -*increment command_line*

To use the **nice** command, specify a value between 0 and 39, indicating how low a priority you want the command to have: 39 is the lowest. If you do not specify a priority, **nice** assumes an increment of 10.

By default, the system processes execute with a nice value of 20. An *increment* value of 15 would cause *command_line* to execute at a nice value of 35, that is, towards the lower end of the priority range, where it would receive proportionately less CPU time.

An *increment* of -10, specified as follows, would increase the priority of the command line to 10:

```
$ nice --10 find / -name chapter6.txt -print > out_file
```

By using **nice --10** to run the command line, you ensure that it takes more processor time than the other programs on the system, thereby increasing its execution speed. Increasing the priority of a job above 20 is available only to the root user.

Note that in the C shell, increasing the nice value of a process works in reverse. For example, to run the **find** command above, you would enter the following:

```
$ nice +10 find / -name chapter6.txt -print > out_file
```

Identifying the niceness of a process

To obtain a listing of your processes and their nice values, enter **ps -l**. The eighth column of output, headed "NI", shows the nice value. The following command line strips out just the command name and the nice value for each of the active processes:

```
$ ps -l | awk '{ printf("%s\t%s\n",$NF,$8) }'
```

(For more information on the **awk**(C) programming language, see Chapter 13, "Using awk" (page 323)). Note that when you run a process in the background under the Korn shell and the **bgnice** variable is set, the process runs with a nice value of 4. (You can examine the Korn shell's internal variables by entering the command **set -o**; you can switch **bgnice** off by typing **set +o bgnice** or switch it on by typing **set -o bgnice**.)

Scheduling your processes

The UNIX system lets you suspend the execution of a command, execute a command at a specific time in the future and execute commands at regular times.

Running processes at some time in the future

You can run commands at an arbitrary time in the future[1] by using the **at**(C) and **batch**(C) commands. **at** allows you to specify a time when the command should be executed, and **batch** executes the command when the system load level permits. The **at** command uses the format **at** *sometime command* where *sometime* is a time and date in the future when *command* will be executed. **at** is useful for sending yourself reminders, for example:

```
$ at 1:00pm Jan 24
mail -s "Technical Publications meeting at 1:15" jdixon
⟨Ctrl⟩D
job 782560800.a at Mon Jan 24 13:00:00 BST 1994
$
```

This command sends a blank mail message entitled Technical Publications meeting at 1:15 to **jdixon** at 1:00P.M. on January 24.

at allows times and dates to be specified in a wide variety of ways. See **at**(C) for details.

To display a list of current **at** jobs, type **at -l**. To remove **at** jobs and their identification numbers, type **at -r** *job_id*; to delete the **at** job queued in the last example, enter the following:

```
$ at -r 782560800.a
```

Note that the trailing ".a" must be specified. This distinguishes **at** jobs from **batch** jobs, which have a trailing ".b".

batch takes no arguments and submits a command for immediate execution at lower priority than an ordinary **at** command. For example:

```
$ batch
compress *.txt
⟨Ctrl⟩D
$
```

---

1. The UNIX system's idea of time starts at the "Epoch", January 1, 1970. The largest number representable by a 32-bit signed integer is 2147483647: add this number of seconds to the Epoch, and you get a date in the year 2037. On systems that implement dates in this way, 2037 is the UNIX system time horizon: representing dates as unsigned 32-bit integers, or even as 64-bit integers, would obviously enhance the UNIX system's grasp of the future.

This command compresses all the files ending in *.txt* in the current directory. The job will be executed when the load on the system permits.

The **at** and **batch** commands are available only to users whose user names appear in the */usr/lib/cron/at.allow* file: users who are specifically barred from using these facilities appear in */usr/lib/cron/at.deny*. These files are editable only by the root user.

Executing processes at regular intervals

The **crontab**(C) command lets you execute routine jobs (called "cron" jobs) on a regular basis. For example, it could be used to periodically back up your files, or to periodically clean up *tmp* and *log* files. To submit a **cron** job, details of the job must be added to a *cronfile*. This is a normal file, but its contents are formatted in a special way:

```
Minutes   Hours   Day_of_Month   Month   Day_of_week   Command
```

Fields are separated by spaces or tabs. The file cannot have blank lines. The *cronfile* parameters are as follows:

Field	Allowable values
Minutes	0-59
Hours	0-23
Day of month	1-31
Month of year	1-12
Day of week	0-6 (0=Sunday)
Command	any non-interactive command

A field can be a number, a range of numbers (for example 10-20), a list of numbers separated by commas, or an asterisk (all values). For example, an asterisk in the "Hours" field means "every hour"; an asterisk in the "Month" field means "every month".

Let us assume that you want to write a *cronfile* to issue reminders and perform regular tasks:

- You need to attend a meeting at 10A.M. every Monday, and you want to remind yourself of this at 9:45A.M. on Monday mornings.

- You want to find and remove any old files beginning with "#" in your home directory at 4:30P.M. on the first day of every month.

- You want to echo the date and time to your terminal at 9:00A.M. Monday to Friday.

Create a file that looks like the following:

```
45 9   *   *   1    echo "Weekly status meeting" > /dev/tty06
30 16  1   *   *    find $HOME -name '#*' -atime +3 -exec rm -f {} \;
0  9   *   *   1-5  echo date > /dev/tty06
```

When you have created your file (called *cronfile*), submit it by typing the following:

```
$ crontab
```

If you want to edit an existing cron job, the *cronfile* should be edited and re-submitted.

To display the current cron job, type **crontab -l**. Redirect the output to a file and edit it, then re-submit the new file. This replaces the old cron job.

To remove the current cron job, type **crontab -r**. If you submit a second *cronfile* before the first one is executed, the first one will be overwritten by the second.

As with **at** and **batch**, access to **crontab** can be turned on and off by the root user by adding user names to */usr/lib/cron/cron.allow* and */usr/lib/cron/cron.deny* respectively.

Delaying the execution of a process

The **sleep**(C) command suspends the execution of a command for a number of seconds. For example **(sleep 20; date > /dev/tty06) &** delays the execution of the **date**(C) command by 20 seconds.

sleep is useful for spacing out commands:

```
$ (while true
> do
> who >> who_report
> sleep 3600
> done)&
$
```

This sequence repeats the **who** command every 3600 seconds (1 hour), and writes the output to a file called *who_report*. (This task could also have been carried out using the **cron** command.)

Using ">>" in the sequence causes the output to be appended to the end of *who_report*. If ">" were used instead, the contents of the file would be overwritten each time the **who** command was run. For an explanation of the **while** structure, see Chapter 11, "Automating frequent tasks" (page 245)).

Chapter 6

Working with DOS

DOS utilities let you access and manipulate files on DOS disks and DOS parti-
tions on your hard disk. This chapter explains how to:

- set up DOS devices under the UNIX system (this page)
- use DOS filenames (page 174)
- list DOS files (page 175)
- copy DOS files between DOS and the SCO OpenServer system (page 175)
- display DOS files (page 176)
- convert DOS files to and from the UNIX system (page 176)
- remove DOS files (page 177)
- create a DOS directory (page 177)
- remove a DOS directory (page 178)
- format a DOS floppy (page 178)
- mount DOS filesystems (page 179)

DOS devices under the UNIX system

The SCO OpenServer system sees every piece of equipment attached to the
computer as a file; it communicates with devices such as mass storage sys-
tems and printers by reading from and writing to special device files stored in
/dev. Each device file has a name that corresponds to the physical attributes of
the device itself. For details of the naming of devices, see "Identifying device
files" (page 182).

173

The hard disk can be divided into one or more *partitions*, each of which is accessed in the same way, using a device file. The DOS partition on the hard disk is accessed through */dev/hd0d* for the first hard disk, and */dev/hd1d* for a second hard disk.

To eliminate the necessity to remember the various device files, a file exists that lets your system administrator define DOS drive names that you can use in place of UNIX device files. This file is called */etc/default/msdos*, and, by default, includes the following entries:

```
A=/dev/install
B=/dev/install1
C=/dev/hd0d
D=/dev/hd1d
```

This means that when you are using the DOS utilities, you can use *A:*, *B:*, *C:*, and *D:* instead of */dev/install*, */dev/install1*, */dev/hd0d*, and */dev/hd1d* respectively.

In addition to the DOS utilities described in this chapter, you can also use the **dd**(C), **diskcp**(C), and **diskcmp**(C) commands to copy and compare DOS floppies, and the **dtype**(C) command to find out what type of floppies you have.

DOS filenames

DOS filenames are all uppercase when viewed from the SCO OpenServer system, and are restricted to eight letters, followed by a period, then a three letter extension. However, when you type the name of a DOS file in conjunction with a DOS utility, you can use upper- or lowercase letters. Because DOS does not provide an equivalent to the "executable" permission bit, the three letter extension is used to indicate if a file is executable. Files ending in .EXE, .SYS or .COM are programs, and files ending in .BAT are batch (script) files.

If you create a file on a SCO OpenServer system and transfer it to a DOS system, try to avoid giving it one of these extensions. Otherwise, DOS may mistake it for an executable program, with unpredictable results.

When copying files from a DOS disk to a SCO OpenServer system, uppercase filenames are automatically converted to lowercase. When copying files from a SCO OpenServer disk to a DOS disk, lowercase names are truncated to fit the DOS filename convention and converted to uppercase.

See **doscmd**(C) for more details of DOS filenaming conventions.

Listing DOS files in standard DOS format

To list the files on a DOS floppy or partition in DOS format, use the **dosdir** command (see **doscmd**(C) for details), as follows:

> **dosdir** *directory*

For example, to list the files on a floppy in the *A:* drive, enter the following command line:

> $ **dosdir a:**

The files are listed in standard DOS format, as follows:

```
CDCONFIG.SYS    1008    10-10-91         7.33p
PROJECT .TXT   80640     1-18-92        12:00a
COLORS  .SYS   17540     1-18-92        12:10a
TREE    .EPS   23565     4-07-92         7:19p
            4 File(s)         851584 bytes free
```

Listing DOS files in a UNIX system format

To list the DOS files on a floppy or DOS partition in UNIX system format, use the **dosls** command (see **doscmd**(C) for details), as follows:

> **dosls** *directory*

For example, to list the files on a floppy in the *A:* drive, type the following:

> $ **dosls a:**

The files are listed in a UNIX system format, for example:

```
CDCONFIG.SYS
PROJECT.TXT
COLORS.SYS
TREE.EPS
```

Copying DOS files between DOS and SCO OpenServer systems

To copy a file, use the **doscp** command (see **doscmd**(C) for details), as follows:

> **doscp** *filename1 filename2*

The *filename1* argument is the source file and *filename2* is the target file; both arguments may include a drive specification and pathname. For example, to copy a DOS file called *PROJECT.TXT* from the *A:* drive to the */tmp* directory, type the following:

> $ **doscp a:project.txt /tmp**

To copy the file back to the floppy, type:

```
$ doscp /tmp/project.txt a:
```

Note that DOS does not recognize links. If you use **doscp** to copy a link to a DOS disk, a complete copy of the file is made. So, if you have two links to the same file called *file1* and *file2*, and copy them to the same DOS disk, the result will be two identical copies of the file, named *file1* and *file2*. The **doscp** command recognizes the standard UNIX system wildcards, so that you can copy groups of files with a single command.

Displaying a DOS file

To display a DOS file, use the **doscat** command (see **doscmd**(C) for details), as follows:

> **doscat** *filename*

For example, to display the *AUTOEXEC.BAT* file on a floppy in drive *A:*, type:

```
$ doscat a:autoexec.bat
```

If you specify more than one file, the files are displayed one after another. If the file or files are too long to fit on a single screen, pipe the output through **more**. For example, to display two files on the *A:* drive called *file1* and *file2*, type the following:

```
$ doscat a:file1 file2 | more
```

Converting DOS files to and from UNIX system file format

Note that DOS text files contain extra formatting characters that will show up on your screen. A line of text in a UNIX system file is terminated by a line feed character. In DOS files, a line is terminated by a line feed and a carriage return (^M). Because no attempt is made to change the nature of DOS files, the carriage return character is visible when editing a DOS file from the UNIX system partition. Thus when a DOS file that contains a series of numbers is opened using **vi**(C), it looks something like this:

```
This is a DOS file.^M
Note that each line ends in a spurious character^M
like this.^M
~
~
~
~
~
~

"TEST.TXT" 3 lines, 100 characters
```

You can either ignore these characters, or remove them with the **dtox**(C) (DOS to UNIX) command. If you remove the carriage returns in a DOS file using **dtox**, you must replace them using the **xtod**(C) (UNIX to DOS) command before you use the file under DOS. The following commands convert the DOS file *test.txt* to and from the UNIX format file *test.out*:

```
$ dtox test.txt > test.out
$ xtod test.out > test.txt
```

Automatic file conversions when using DOS utilities

When you display a DOS file using **doscat** or copy a DOS file to your SCO OpenServer system using **doscp**, the carriage return characters (^M) are automatically stripped out of the file. When text files are transferred to DOS, the commands insert a ^M before each linefeed character.

Under some circumstances, however, the automatic newline conversions do not occur. **doscat** and **doscp** supply the **-m** option, which ensures that the newline conversion is carried out. The **-r** option overrides the automatic conversion and forces the command to perform a true byte copy regardless of file type.

Removing a DOS file

To remove a file from a DOS floppy or partition, use the **dosrm** command (see **doscmd**(C) for details), as follows:

dosrm *filename*

For example, to remove a file called *TEST.TXT* from the *A:* drive, type the following:

```
$ dosrm a:test.txt
```

Creating a DOS directory

To create a DOS directory, use the **dosmkdir** command (see **doscmd**(C) for details), as follows:

dosmkdir *directory*

For example, to create a directory called *PROJECTS* on the *A:* drive, execute the following command line:

```
$ dosmkdir a:projects
```

Removing a DOS directory

To remove an empty DOS directory, use the **dosrmdir** command (see **doscmd**(C) for details), as follows:

> **dosrmdir** *directory*

For example, to remove a directory called *PROJECTS* from the *A:* drive, type the following:

```
$ dosrmdir a:projects
```

Before you remove a directory, you must make sure that it does not contain any files or subdirectories. If it is not empty, you must either delete the files it contains (using **dosrm**), or move them to other directories (using **doscp** to copy the files to a different directory, followed by **dosrm** to remove the files from the directory to be removed).

The following is a short script to remove all the files from a DOS directory:

```
#!/bin/sh
for target in `dosls $1`
do
      dosrm $1/${target}
done
```

For example, if the script is named *dosdirempty*, and your disk in drive *A:* contains a directory called *work*, you would type the following in order to remove all the contents of the *work* directory:

```
$ dosdirempty A:WORK
```

Formatting a DOS floppy

DOS partitions on your hard disk cannot be formatted using the procedures described in this section. To do this, you must either use the native DOS **format** command, or have root user privilege (see Chapter 7, "Using other operating systems with an SCO system" in the *SCO OpenServer Handbook* for details).

Although you cannot format a hard disk, you can format a DOS floppy disk using the **dosformat** command (see **doscmd**(C) for details), as follows:

> **dosformat** *drive*

For example, to format a floppy in the *A:* drive, type the following command:

```
$ dosformat /dev/fd0
```

Note that you cannot specify *A:* (or *a:*) with **dosformat**. This is because *A:* is aliased to */dev/install*, which cannot be formatted (see "DOS devices under the UNIX system" (page 173)). You should therefore use */dev/fd0*, which automatically formats the floppy correctly for the drive type. (You can specify the

complete device filename if you wish, for example, */dev/fd096ds15* for a 5¼ inch 1.2MB drive; see "Identifying device files" (page 182).)

Note that unlike the **format**(C) command, that only works on raw devices (such as */dev/rfd0*), **dosformat** only works on block devices (such as */dev/fd0*).

You can format a floppy in the second drive by typing either of the following command lines:

    $ dosformat /dev/fd1

    $ dosformat /dev/fd196ds15

The latter formats a 5¼ inch 1.2MB drive.

Using mounted DOS filesystems

In addition to using the DOS utilities, you can mount a DOS filesystem and access its files directly while still operating from the SCO OpenServer system. A general description of mounting filesystems is given in "Mounting a filesystem" (page 98).

This means that you can edit DOS files in place, without first copying them into the UNIX filesystem. The SCO OpenServer system deals with DOS files by superimposing certain qualities of UNIX filesystems over the DOS filesystem without changing the actual files. UNIX filesystems are highly structured and operate in a multiuser environment. In order to make DOS files readily accessible, access permissions and file ownership are superimposed on the DOS filesystem when mounted.

The major restriction with mounting a DOS floppy or a DOS partition is that DOS applications (for example, your DOS word processing package) cannot be executed under this arrangement.

If you need to use your DOS applications, you (or your system administrator) need to do one of the following:

- Boot the system as a DOS system (only your system administrator can do this).

- Make the DOS partition the active partition, so that the computer boots DOS by default (only your system administrator can do this).

- Run SCO® Merge™ from the UNIX partition. (Ask your system administrator if you have this application on your system.)

DOS utilities cannot be used on a mounted DOS filesystem. Normally, only your system administrator can mount a filesystem. Access by users is governed by the permissions and ownership that your system administrator places on the DOS filesystem. The system administrator must either mount the

DOS filesystem or set up the system so that users can use the **mnt**(C) command. The filesystem must also be mountable. Such systems have an entry in */etc/default/filesys* that contains the command `mount=yes`. See "Mounting a filesystem" (page 98) for details of */etc/default/filesys* and how to check its contents.

Because of the limitations discussed earlier, DOS does not recognize permissions or ownership. When mounted from the UNIX partition, DOS files behave as follows:

- The permissions and ownership of the filesystem are governed by the mount point. For example, if *root* creates a mount point */X* with permissions of 777, all users can read or write the contents of the filesystem. If *root* creates a mount point with permissions of 700, only *root* can read or write the contents of the filesystem.

- The ownership of regular files is governed by the mount point. If the mount point is owned by *root*, all files within the DOS filesystem and any created by other users are all owned by *root*.

- DOS does not distinguish between users, therefore files can be either readable/writable for everyone, or read-only for everyone. The permissions for regular files are either 0777 for readable/writable files or 0555 for read-only files. When a file is created, the permissions are based on the **umask** of the creator, but may not be the same as they would be in a normal filesystem. If the **umask** allows anyone to write to the file, then the permissions are set to 777; if no one can write to the file, the permissions are set to 555. See **umask**(C) for details.

Points to note when using files on a mounted DOS filesystem

The rules for filenames and their conversion follow the guidelines found in **doscmd**(C). Filenames that exceed the limit for a DOS filename will be truncated when you copy them to a DOS partition. For example, if you attempt to create a file named *rumpelstiltskin* within the DOS filesystem, it is truncated to *rumpelst*.

You can use wildcard characters just as you use them with UNIX filesystems.

When accessed from the UNIX partition, the creation, modification, and access times of DOS files are always identical and use GMT, or Greenwich Mean Time. (This is because UNIX systems use GMT internally and convert it for you.) This means that files created in the DOS filesystem will not have consistent times across the operating systems.

You cannot use the **backup**(ADM) utility to make backups of a mounted DOS filesystem. However, DOS utilities and other copy programs like **tar**(C) work as expected.

Chapter 7

Working with disks, tapes, and CD-ROMs

Most of the time you work with files, they are stored on your computer's hard disk and are accessible using the file and directory handling commands discussed in Chapter 3, "Working with files and directories" (page 79). However, it may be necessary to use other types of storage device. For example, you might want to pass a copy to the user of another machine not connected to your own, or to store infrequently used material, or to make a backup copy of your work. This chapter explains how to use:

- devices (this page)
- tapes (page 185)
- CD-ROMs (page 187)
- the **tar** archiving command (page 187)
- the **cpio** archiving command (page 190)

Using UNIX devices

The SCO OpenServer system sees every piece of equipment attached to the computer as a file; it communicates with it by reading from and writing to a special *device file* located in the */dev* directory. Each type of device has its own device file, which in turn points to a piece of software called a *device driver*, which is linked into the kernel.

A huge number of device drivers have been written for UNIX systems. One way of controlling the "footprint" of a UNIX implementation (the amount of memory taken up by the system) is to limit the number of device drivers in the kernel: for example, a normal office system is unlikely to require the driver for a barcode reader. There are, however, device drivers that can be loaded into the system when needed. The SCO OpenServer system supports "boot-time loadable drivers" (BTLD's): these are described in "Using boot-time loadable drivers" in the *SCO OpenServer Handbook*.

Two types of device file are supplied in */dev*, those for the *character* devices (also known as "raw" devices), and those for the *block* devices (also known as "buffered"). The former handle input/output operations of arbitrary size while the latter operate through fixed size buffers.

Some devices may supply both types of interface, and thus have two separate device files in */dev*. Typically, disk devices supply both, and the following entries will appear in a long listing:

```
$ ls -l
brw-rw-rw-   6 bin      bin      2, 52 Mar 24  1993 fd096ds15
crw-rw-rw-   5 bin      bin      2, 52 Mar 24  1993 rfd096ds15
```

In this case, the one floppy disk drive supplies both a block interface, *fd096ds15*, and a raw interface, *rfd096ds15*. Note also the first character position of the listing: the "b" and "c" indicate "block" and "character" respectively. See **ls**(C) for more details of file types; for more information on devices, refer to the *SCO OpenServer Handbook*.

Identifying device files

The device files in */dev* have names that correspond to the characteristics of the devices themselves. The following steps show how to work out the name of the device file to use when specifying a floppy disk drive, for example:

1. All floppy disk devices are located in */dev* and begin with *rfd* (the "r" is short for "raw", because the SCO OpenServer system has to access the disk directly). The "fd" stands for floppy disk. So start the device filename with */dev/rfd*.

2. If your computer has only one floppy disk drive, follow this with a number "0". If your computer has two or more drives, you can follow it with a "0", "1" or higher number (depending on whether you want to format a disk in the first, second, or subsequent drive).

3. Follow this digit with the number of tracks per inch on the disk. This is usually indicated on the disk label. The standard number of tracks per inch on different disks are as follows:

   48 Low-density 5¼ inch disks (360KB)

   72 Double-density 3½ inch disks (720KB)

   96 High-density 5¼ inch disks (1.2MB)

   135 High-density 3½ inch disks (1.44MB)

4. Follow this number with either "ds" if the disk is double-sided, or "ss" if it is single-sided.

5. Finally, finish the device name by adding the number of sectors per track on the disk, as follows:

   9   Standard 5¼ inch or 3½ inch floppy disk

   15 High-density 5¼ inch floppy disk

   18 High-density 3½ inch floppy disk

In this way, a name such as */dev/rfd096ds15* can be constructed to indicate a floppy disk drive. Its name literally means "raw floppy disk 0; 96 tracks per inch; double-sided; fifteen sectors per track."

A summary of the most common types of floppy disk is given in the following table:

Device code	Size and type of backup media
rfd048ds9	5¼ inch 360KB floppy disk
rfd096ds15	5¼ inch 1.2MB floppy disk
rfd0135ds9	3½ inch 720KB floppy disk
rfd0135ds18	3½ inch 1.44MB floppy disk

Note that all of these entries apply to the first drive (drive 0) on the system. If you want to use the second drive, change the 0 after "rfd" to 1; if you want to use the third drive, change the 0 to 2, and so on.

Default devices

Many commands, such as **format**(C) and **tar**(C), have a default device, which is accessed automatically whenever you invoke the command without specifying a device. This default device is listed in the files in the directory */etc/default*. The file */etc/default/format*, for example, which specifies the default device used by the **format** command, contains the following:

```
#        @(#) format135 23.2 91/08/29
    .
    .
    .

VERIFY=Y
DEVICE=/dev/rfd096ds15
```

Here, the default device is the floppy disk drive */dev/rfd096ds15*. The VERIFY line indicates that floppy disks are to be verified after they are formatted, to ensure that the process was successful.

Using floppy disk drives

In addition to using floppy disks to store "loose" files, you can also use them to back up an entire system; this may, however, require the availability of many disks. In either case, a floppy disk must be formatted before it can be used.

Formatting floppy disks

Before a new floppy disk can be used it must be formatted. Formatting is a one-time process that writes essential information on the surface of the disk. If you format a disk that already contains information, you will destroy the previous contents of the disk. To format a floppy disk, ensure that it is in the appropriate drive, ensure that it is writable, and then use the following command:

> **format** [*drive*]

The *drive* argument is an optional *device file* to use if you want to format a device other than the default one named in */etc/default/format*. (See "Identifying device files" (page 182) for an explanation of how to identify the device file to use for a given disk drive.) Note that only raw devices can be formatted.

For example, to format a high-density 3½ inch double-sided floppy disk in the second disk drive, type the following:

```
$ format /dev/rfd1135ds18
```

The **format**(C) command will print out a brief status message and ask if you want to continue. If you have made a mistake, you should type **n** at this point.

See also "Formatting a DOS floppy" (page 178).

Determining how many disks you need for a backup

You can back up a whole system onto floppy disks: to estimate how many floppy disks the backup requires, use the **du**(C) disk usage command, which returns the number of 512-byte blocks contained (recursively) within the current directory. Its **-s** option prints only a grand total:

```
$ du -s
24356   .
```

For example, if 24356 is displayed, this means that you require a total of 24356 x 512 bytes, or roughly 11.9MB. This would give a total of nine 1.4MB floppy disks to create a full backup. You can display the number of blocks for each individual file or directory by including the filename on the end of the **du** command, as in the following:

```
$ du -s /dev
46      /dev
```

Once you know how many floppy disks are required, you can use the **tar**(C) or **cpio**(C) commands to create the archive. See "Creating a backup with tar" (page 187) and "Creating a backup with cpio" (page 190) for details of how to use these commands.

Using tapes

You can copy files to and from tape devices in the same way as you do with floppy disks. However, there are a number of differences between tapes and floppy disk systems. Notably, although magnetic tapes can store far more data than a floppy disk, they can only provide *serial access* to the information; that is, when reading or writing a tape, you must start at the beginning and read through each file until you get to the end: you cannot jump around or skip files.

To copy files to and from a tape device, you should use **tar** or **cpio**, with the appropriate device file (from the list below). You may also need to use the **tape** command to control the tape drive directly; see "Rewinding, erasing, and retensioning tapes" (page 186).

There are several different types of tape that may be available on your machine. The following are the most common:

QIC-02   A full-sized quarter-inch tape cartridge; the first QIC-02 drive uses the */dev/rct0* device file. After accessing the tape, this device automatically rewinds the tape. If you store more than one archive on the tape, you must use the no-rewind device file */dev/nrct0* to access the second and subsequent files.

QIC-40/QIC-80   Smaller mini-cartridge units related to the QIC-02 format. These devices are accessed through the */dev/ft0* device file.

Mini-cartridge   Mini-cartridge tape drives linked to the floppy disk drive controller. These differ significantly from the QIC family of tape drives. Notably, you must format mini-cartridge tapes before using them (see "Formatting tapes" (this page)). Mini-cartridge drives are accessed via the */dev/ctmini* device file.

SCSI   SCSI tape drives are controlled by a SCSI controller; they are accessed via the devices named */dev/Stp0, /dev/Stp1*, and so on.

Formatting tapes

Most tapes do not require formatting as this is done during manufacturing. Mini-cartridge tapes do, however, require you to format them, using the **format** option to the **mcart**(C) command, as follows:

mcart format [*device*]

The default device file is */etc/default/mcconfig*: to format a tape in any other drive, supply the *device* argument.

Rewinding, erasing, and retensioning tapes

To rewind or erase a tape, you should use the **tape**(C) command.

To rewind a tape, the command is as follows:

tape rewind *device*

The *device* argument is the name of the device file for the tape unit. (See "Using tapes" (page 185).) For example, with a quarter-inch QIC-02 tape, enter the following command line:

```
$ tape rewind /dev/rct0
```

By default, if no device is specified, **tape** reads the file */etc/default/tape*. This contains the device name of a tape drive to use. So if there is only one tape device on your system and */etc/default/tape* is correctly set up, you should not need to specify a device name.

It is a good idea to rewind the tape to the beginning after every use, or after encountering an error.

To erase a tape, the command is as follows:

> **tape erase** *device*

You should retension any tapes that you use regularly, or that have been in storage and that you now wish to read from; this takes up any slack in the cartridge and reduces the likelihood of errors. (The drive retensions a tape by winding to the end of it, then rewinding it rapidly.) The command to retension a cartridge is as follows:

> **tape reten** *device*

We recommend that you write-disable your tapes to prevent accidental erasure or overwriting. This is done by turning the tab on the cartridge so that the arrow points to the SAFE position; turn it the other way when you intend to write over or erase the tape.

Using CD-ROMs

A CD-ROM is a special kind of compact disk that stores computer data rather than digital audio data. CD-ROMs differ from floppy disks or tapes in that they are *read-only* media; you can read data stored on them, but you cannot write data onto a CD-ROM. Most CD-ROMs are used as large data repositories, as they typically store 300MB or more. For example, the entire SCO OpenServer system fits on a single CD-ROM with room to spare, but takes as many as 60 floppy disks.

CD-ROMs are preformatted, and the data is stored on them in the form of a special type of filesystem (defined by the High Sierra or ISO 9660 standards) that can be mounted onto the SCO OpenServer system. For details of mounting a filesystem, see "Mounting a filesystem" (page 98).

Creating a backup with tar

A **tar**(C) backup is a special file that contains other files and their associated directory information in linear order. **tar** was originally created for archiving files to tape, hence the name.

You can specify a list of files to archive by name; and copies of all of the files will be stored in the archive. **tar** creates the archive by reading the files one at a time, writing a header (containing information about the files) and then writing the contents of the files to the device or tarfile. Consequently, a tarfile is always slightly larger than the total size of the files it contains.

You may find **tar** useful for archiving infrequently-used files in a directory; create a tarfile containing copies of all the files, remove the originals, then use **compress**(C) to reduce the size of the tarfile. The result is a convenient, compact package containing the archived material.

To create a **tar** file on a floppy disk, use the command **tar cv*n*** where the **c** option creates a new backup, overwriting any data already existing on the *device*, **v** (verbose) displays each file as it is copied, and *n* is the number of a device specified in the "Key" column of the file */etc/default/tar*. To see the list of available device numbers, type **tar** without arguments, as follows:

```
$ tar
Usage: tar -{txruc}[0-9vfbkelmnpwAFL] [tapefile] [blocksize]
        [tapesize] files ...
        Key     Device          Block   Size(K)   Tape
        0       /dev/rfd048ds9    18      360       No
        1       /dev/rfd148ds9    18      360       No
        2       /dev/rfd096ds15   10      1200      No
        3       /dev/rfd196ds15   10      1200      No
        4       /dev/rfd0135ds9   18      720       No
        5       /dev/rfd1135ds9   18      720       No
        6       /dev/rfd0135ds18  18      1440      No
        7       /dev/rfd1135ds18  18      1440      No
        8       /dev/rct0         20      0         Yes
        9       /dev/rctmini      20      0         Yes
```

Using this list, you can select the size of floppy disk you want to use. For example, to create a tarfile on a 720KB floppy disk in the second floppy disk drive, containing all files in the directory *work*, type the following command line:

```
$ tar cv5 work
```

For details of how floppy disk drive names are constructed, see "Identifying device files" (page 182).

To use a device not listed in */etc/default/tar*, use the **tar f** option. This causes the next argument (*device*) to be the target drive. For example, the command to back up all the files from the current directory and all subdirectories to a 1.2MB floppy in drive 0 is as follows:

```
$ tar cvf /dev/rfd096ds15 .
```

In this example, **tar** uses a relative pathname, (.), standing for the current working directory, rather than an absolute pathname. This allows you to restore the files to another location. If you give **tar** absolute pathnames, it will restore files to their original location, creating any intervening directories.

If you want to add files to an archive, rather than overwriting the existing contents, use the **r** option instead of **c**; this appends the files to the archive rather than creating a new one. Note that this only works for archives held on disk; you cannot use the **r** option with tapes.

If you do not specify a device, the **tar** backup will be created on the default device defined in */etc/default/tar*. This default entry is not displayed when you type **tar** on the command line; list the contents of */etc/default/tar* to see something like the following:

```
$ cat /etc/default/tar
   .
   .
   .
# Default device in the absence of a numeric or "-f device" argument
archive=/dev/rfd0135ds18        18      1440     n
```

Note that no "Key" column is displayed.

If you specify a filename instead of a device name, **tar** will still create the backup. For example, you could store the files from *work* in a file called *work.tar* in the current directory with the following command:

```
$ tar cvf work.tar work
```

Listing the files in a tar backup

To see a list of the contents of a **tar** backup, use the following command:

tar tvf *device*

The *device* argument is either the name of the device where the backup is stored, or the name of the file containing the backup. The **t** and **v** options combine to display a directory listing, the output of which is similar to **ls -l**. The **f** option tells **tar** to use the next argument (*device*) as the backup to read from; for example:

```
$ tar tv2
tar: blocksize=20
rw-r--r--13079/1014    713 Dec 17 11:10 1992 READ.ME
rw-rw-r--13079/1014     77 Dec 17 09:33 1992 00.partno.nr.Z
rw-rw-r--13079/1014   1887 Dec 17 09:33 1992 00.title.nr.Z
rw-rw-r--13079/1014   8735 Dec 17 09:36 1992 00.toc.nr.Z
rw-rw-r--13079/1014   5961 Dec 17 09:37 1992 01.intro.nr.Z
```

Extracting files from a tar backup

To extract files from a **tar** backup, the format is as follows:

tar xvf *device files*

tar looks at the backup on *device* (or the file of that name), extracts all the files which match *files*, and places them relative to the current directory. Note that **tar** does not understand wildcards so *files* should be a list of explicit filenames.

```
$ tar xvf /dev/rfd096ds15
```

This command line restores the contents of the backup as held on */dev/rfd096ds15*. The **x** option tells **tar** to extract files, **v** displays them, and **f** tells **tar** to use the next argument (*/dev/rfd096ds15*) as the name of the backup.

(If the use of wildcards is especially important, you should use the **cpio**(C) command instead of **tar**; see "Creating a backup with cpio" (this page) for details.)

If, when you created the backup, you gave **tar** a full pathname, the files will be restored to their original location. If you gave **tar** a relative pathname, you can restore the files to a different location.

You can tell whether the files in a **tar** archive have full pathnames by listing the files in the archive; if their names begin with a slash (/) they will be extracted relative to the root directory. For example, the files in the following archive will be placed in the root directory (unlike the files in the previous example):

```
$ tar tvf /dev/rfd096ds15
rw-r--r--13079/1014     713 Dec 17 11:10 1992 /README
rw-rw-r--13079/1014      77 Dec 17 09:33 1992 /00.partno.nr.Z
rw-rw-r--13079/1014    1887 Dec 17 09:33 1992 /00.title.nr.Z
rw-rw-r--13079/1014    8735 Dec 17 09:36 1992 /00.toc.nr.Z
rw-rw-r--13079/1014    5961 Dec 17 09:37 1992 /01.intro.nr.Z
rw-rw-r--13079/1014   61179 Dec 17 09:44 1992 /02.op.nr.Z
```

You can override full pathnames by specifying the **A** option, which makes **tar** write the files as though your current directory is the root directory. For example, the file */README* (as above) is normally unpacked and placed in the root directory; but if you restore it by using the command **tar xvA** while your current directory is */u/charles* the file will be placed in */u/charles/README*.

Creating a backup with cpio

The **cpio**(C) command is a more sophisticated backup tool than **tar**. It is harder to use, but is capable of copying special files (such as devices and links) consistently, and will accept wildcard characters when listing the files to be archived.

To create a **cpio** (copy in/out) backup, use the **-o** (output) mode. You feed a list of files to **cpio**'s standard input; for example, by piping the output from **ls** to **cpio**. **cpio** then copies the files into a single **cpio** backup file on its standard output, which should be redirected to the appropriate backup device.

For example, to copy all the regular files below your current directory to a 1.2MB disk in the first floppy drive, type the following command line:

```
$ find . -type f -depth -print | cpio -ocv > /dev/rfd096ds15
```

This command uses **find**(C) to locate all the regular files (**-type**) in your current directory (.) and its subdirectories (**-depth**), printing their names through a pipe (|) to **cpio**. It outputs the names of entries in a directory before the directory itself; that is, it searches "depth first," going to the bottom of the filesystem. **cpio** then outputs (**-o**) those files into an archive on */dev/rfd096ds15*, a high-density 5¼ inch floppy disk. As it does this, **cpio** displays their names on the terminal (**-v**).

The **-c** option writes the header information in ASCII format. This is a special option that allows **cpio** archives to be read on any other type of machine equipped with **cpio**, even if the target machine's architecture differs radically from that of the computer on which the archive was written. (Otherwise, the archive might be nonportable, even though both machines could be running the SCO OpenServer system and **cpio**.)

The **find** command above used a relative pathname (.) rather than an absolute pathname. This allows you to restore the files to another location. If you give **cpio** absolute pathnames, files will be restored to their original location, overwriting any existing data (as with **tar**).

You may need to make a backup that is larger than the capacity of the disk or tape being used. If this is the case, **cpio** will stop when it fills the backup medium, and will write a short message prompting you to insert another disk (or tape) and type the name of the device to continue using. If you want to continue backing up on the same device, replace the media and type the device name; if you simply press ⟨Enter⟩, **cpio** will terminate.

Pass mode (**cpio -p**) works like output (**cpio -o**) mode, except that it copies files to another directory in the filesystem. You can back up files to another floppy disk or to a remote filesystem mounted on your system. You can then restore these files with **cpio**. For example, to copy all files from your current directory (including all its subdirectories) to a remote filesystem, mounted on the */mnt* directory on your system, use the following command:

```
$ find . -depth -print | cpio -pvd /mnt/backup6
```

In this example, *backup6* is the name of the backup subdirectory on the */mnt* filesystem. This command starts **find** in the current directory. Rather than following its normal search order (which is to scan all the files in the current directory before entering subdirectories), **find** with the **-depth** option dives down as deep as possible in the directory tree, then lists all the files it encounters as it searches. Because this option lists directory paths before the files stored in them, **cpio** creates the directories before the files.

Listing the files in a cpio backup

To list the contents of a **cpio** backup, use the following command:

   cpio -itv < *device*

The *device* argument is either the name of the device where the backup is stored, or the name of the file containing the backup. The **-i** option defines **cpio** input mode, and **t** and **v** combine to display a directory listing, the output of which contains the same information as **ls -l** (although the format differs slightly):

```
$ cpio -itv <archive.cpio
100644 charles       693  Jan 29 16:20:03 1993  book.order
100644 charles      3783  Feb 01 13:36:34 1993  checkmac.out
100644 charles     16457  Jan 26 14:02:59 1993  command.eps
100644 charles      1199  Jan 29 10:36:04 1993  filesched
100644 charles     21064  Jan 26 14:03:00 1993  permis.eps
100644 charles     29954  Jan 26 14:03:01 1993  procTree.eps
100644 charles       422  Jan 28 15:17:26 1993  review.log
143 blocks
```

Extracting files from a cpio backup

To extract files from a **cpio** backup use the **-i** (input) mode:

   cpio -i *options* < *device files*

cpio reads its input from the backup on *device* and extracts anything which matches the specified *files*. If, when you created the backup, you gave **cpio** a full pathname, the files will restore to their original location. If you gave **cpio** a relative pathname, you can restore the files to a different location. The following command restores the contents of the backup from device */dev/rfd096ds15* to your current directory:

```
$ cpio -ivcd < /dev/rfd096ds15
```

The **-v** option displays the files as they are restored, **-c** tells **cpio** that there is a header, and **-d** creates subdirectories as needed.

Chapter 8

Using UUCP and dialup commands

You can exchange files and electronic mail with other systems via UUCP (UNIX to UNIX Copy). UUCP is a point-to-point protocol designed for communicating over telephone and serial lines, in contrast to newer networking systems like TCP/IP, that are optimized for communications over local area networks.

If your computer is connected to a larger network, you are unlikely to need UUCP. However, UUCP is still widely used for transporting electronic mail between computers that do not have TCP/IP, and is especially useful for isolated systems that obtain their e-mail feed via a modem.

The UUCP programs allow you to transfer files between remote computers and to execute commands on remote computers. Since the computers may be connected by telephone lines, UUCP transfers can take place over thousands of miles. A UUCP site in New York City can transfer a file to, or execute a command on, a connected UUCP system site in San Francisco, or anywhere in the world.

Note that the UUCP commands do not allow you to have an interactive session with the remote site. If you want to have an interactive session, use the commands discussed in "Using two computers at the same time" (page 202).

This chapter describes how to:

- transfer files between UNIX systems (page 194)
- execute commands on remote UNIX systems (page 200)
- dial up remote systems (page 201).

Transferring files between UNIX systems

Transferring copies of binary and text files between remote UUCP systems can be achieved using the **uucp**(C) and **uuto**(C) commands. There are advantages and disadvantages to each. The **uucp** command gives you great flexibility in specifying where on the remote system the transferred file is to be placed, but the **uucp** command line can be complicated. The **uuto** command, on the other hand, is easy to use, but restricts where you can place the file on the remote system. In addition, retrieving a file sent with **uuto** is slightly more complicated than retrieving a file sent with **uucp**.

Before you can copy files to remote sites with **uucp**, you must verify that:

- Your site knows how to call the remote site.

- The files that you want to send have read permission set for others.

- The directory that contains the file that you want to send has read and execute permissions set for others. This allows the directory to be searched.

- Your computer has write permission in the directory on the remote site to which you want to copy the file.

You must be sure that your computer "talks" to the site with which you want to communicate. The **uuname** command gives you this information. Entering **uuname** with no options lists the UUCP sites your computer talks to directly. For example:

```
$ uuname
kate
rachel
$
```

Entering **uuname** with the -l option causes the name of your computer to be displayed. For example:

```
$ uuname -l
jane
$
```

This tells you the name by which your computer is known on the UUCP network.

Note that you might be able to communicate with a site that does not show up in a **uuname** listing. This is possible because UUCP sites are often "chained together". So if you know that a site to which you want to transfer files communicates with a site your system communicates with, you can send files to the first site through the second.

For example, say your system (called *jane*) knows how to communicate with two other systems (called *kate* and *rachel*, as in the previous example). If *kate* has connection to a fourth system, *alice*, you can send files to *alice* via *kate* by typing **kate!alice** to specify a path. This is equivalent to the mail addressing technique described in the *Mail and Messaging Guide*.

Finally, you must verify that your computer has write permission on the directory on the remote site to which you want to transfer files. Each remote UUCP site has a */usr/lib/uucp/Permissions* file. This file specifies the directories on that site from which your computer can read and to which your computer can write. You can only send a file to a directory on a remote site if your computer has write permissions on that directory, as specified on the remote site's */usr/lib/uucp/Permissions* file.

In order to copy a file to a remote UUCP site, the file must have read permission set for others and the directory that contains the file must have read and execute permissions set for others. Use the **l** command to examine the file's permissions and the **l -d** command to examine the directory's permissions. If the permissions are not correct and you own the file, enter the following commands to set the correct permissions:

> **chmod o+r** *filename*
> **chmod o+rx** *directory*

(See **chmod**(C) for details.) By default, most UUCP sites permit calling-in computers to write to their */usr/spool/uucppublic* directory, which is available on most UUCP systems as a general purpose file transfer directory. Since there is no way to find out which directories your computer can write to on the remote site, other than contacting somebody at the site, the safest thing to do when making a UUCP transfer is to write to */usr/spool/uucppublic*. The procedure for doing this is outlined below.

Using the uucp command

The **uucp** command is equivalent to the ordinary **cp** command, but it transfers files *between* machines. Thus, in addition to specifying a source filename and a destination filename, you need to specify the name of the computer on which the source file is located and the name of the destination computer.

For example, to copy the file *testfile* from machine *kate* to machine *rachel*, if the appropriate permissions are set, you can use the command:

> **uucp kate!testfile rachel!testfile**

The exclamation point (!) separates the host (computer) name from the filename.

Note that, as the exclamation point has special meaning to the C shell, if you are using the C shell you must "escape" with a backslash (\) any exclamation points that appear in a **uucp** command. For a C-shell user, the command above is specified as:

uucp kate\!testfile rachel\!testfile

The format of the **uucp**(C) command is:

uucp [*options*] *src_computer*!*src_file dest_computer*!*dest_file*

where:

src_file   Is the name of the file that you want to copy.

src_computer   Is the name of the computer on which *src_file* is located.

dest_file   Is the name of the copied file on the receiving computer. Usually, *src_file* and *dest_file* are the same.

dest_computer Is the name of the computer on which *dest_file* is located.

options   Are the optional arguments to the command. Options include:

          -j   Prints a job identification number. (The job ID can be used to cancel a pending **uucp** command.)

          -m   Mails the requester when the operation has been carried out.

          -n*user*   Notifies *user* (a username on the destination machine) that a file was sent.

There are several ways to specify the location on the remote machine to which you want to transfer the file. The simplest is the *~/dest_file* specification. This is also the safest specification, because *~/dest_file* is expanded to */usr/spool/uucppublic/dest_file*, ensuring that the transfer will succeed.

For example, to send */usr/andrew/transfile* on your machine *kate* to */usr/spool/uucppublic* on machine *rachel*, enter the following command:

uucp /usr/andrew/transfile rachel!~/transfile

This command creates the file */usr/spool/uucppublic/transfile* on *rachel*.

If */usr/andrew* is your current directory, you can copy *transfile* to *alice* with the following command:

uucp transfile alice!~/transfile

Another form of the command allows you to specify the full pathname of the copied file on the remote computer. However, you must be sure that your computer has write permission on this directory, otherwise the transfer will fail.

As an example, suppose that you want to send */usr/andrew/transfile* on machine *jane* to the */usr/andrew* directory on *kate*. To do so, enter the following command:

> **uucp jane!/usr/andrew/transfile kate!/usr/andrew/transfile**

The **uucp** command can be used to retrieve files from a remote site, in addition to copying files to a remote site. Using the example above, if you are in the */usr/john* directory on *jane* and you want to retrieve a copy of */usr/andrew/transfile* from *kate*, enter the following command:

> **uucp kate!/usr/andrew/transfile   /usr/john/transfile**

You can also use *~user* to specify a location on the remote computer. The *~user* argument is expanded to the pathname of *user*'s home directory on the remote computer. For example, suppose */usr/john* is the home directory of account *john* on machine *kate*. To copy *transfile* from */usr/andrew* on *jane* to */usr/john* on *kate*, enter the following command:

> **uucp /usr/andrew/transfile kate!~john/transfile**

The receiving computer expands *~john* to the full pathname of *john*'s home directory, creating */usr/john/transfile*. Again, your computer must have write permission in *john*'s home directory in order for this transfer to succeed.

With the **uucp** command, files are not copied and sent immediately. Instead, copies are placed in a spool directory and sent once the **uucico** daemon awakens. **uucico** dials remote hosts and transfers the files that it finds queued. Depending on how your system is configured, a **uucp** transfer might take place within minutes or within hours. It is common for **uucp** transfers via modem to be saved up for transmission while cheap rate telephone calls are in effect; this is done via a system **cron**(C) job.

Transferring files to systems that are not connected directly to your own

You might be able to send files to a UUCP site that is not connected directly to your own, that is not listed in a **uuname** listing. For example, your local computer *jane* is connected to a UUCP site named *kate*. *kate* is connected to a UUCP site named *alice*. You can send */tmp/transfile* on your local computer through *jane* to */usr/spool/uucppublic* on *alice*. To do this, specify the full UUCP address relative to your local computer:

> **uucp /tmp/transfile jane!alice!~/transfile**

Note that each site name in the command line is followed by an exclamation mark. By placing several site names in a **uucp** command line, you can greatly extend the range of systems to which you can copy files with **uucp**. This is also true for the **uuto** and **uux** commands discussed below.

8

Checking the status of pending file transfers

You can use the **uustat**(C) command to check on the status of files copied with **uucp**. To check on the status of all your **uucp** jobs, enter **uustat** to display the following output:

```
1234 2/19-10:29 S machine2 mike 9 transfer.file
```

The fields are as follows:

1234	The job number assigned to this **uucp** transfer. (This number is unique to **uucp** and is used to keep track of the queue of files awaiting dispatch).
2/19-10:29	The date and time the job was queued in the spool directory.
S	The status of the job: "S" indicates the job is a file to be sent; "R" indicates the job is a request.
machine2	The site name of the recipient's computer.
mike	The login of the user who requested the transfer.
9	The size of the job in kilobytes.
transfer.file	The name of the file.

If the transfer is completed, **uustat** displays the message:

```
COPY FINISHED, JOB DELETED.
```

Several options are available to use with **uustat**. Refer to **uustat**(C) for more information.

Transferring files to a public directory using uuto

The **uuto** command allows you to copy files to the public directory of a UUCP site to which your system is connected. The public directory on most UNIX and XENIX® systems is */usr/spool/uucppublic*. For example, to send a file from the machine *jane* to the machine *kate*, use:

> **uuto mytextfile kate!john**

john is the login of the user to whom you are sending files; *john* can retrieve *mytextfile*, as explained in "Retrieving files from the public directory" (page 199).

Before you can send a file with **uuto**, you must verify that:

- the file has read permission set for others

- the directory that contains the file has read and execute permissions set for others

If the permissions are not correct, enter the following commands to set the correct permissions. For example:

chmod o+r mytextfile
chmod o+rx .

(The dot (.) stands for the current directory.) Files sent with **uuto** are placed in the directory:

*/usr/spool/uucppublic/receive/**login**/**src_computer***

where *login* is the login of the user to whom you are sending files and *src_computer* is the site name of your system. In the example above, *mytextfile* is placed in */usr/spool/uucppublic/receive/john/jane*.

As an example, suppose that you want to send a copy of *transfile* in */tmp* on *jane* to the user *sara* on the machine *kate*. To do so, enter the following command:

uuto /tmp/transfile kate!sara

This command copies *transfile* to the following directory:

usr/spool/uucppublic/receive/sara/jane

When the file transfer is complete, the recipient is notified by **mail** that the file has arrived. If the **-m** option is used on the **uuto** command line, the sender is notified by **mail** of the success or failure of the transfer.

Like **uucp**, files transferred with **uuto** are not transferred immediately after the command is entered. Instead, they are placed in a spool directory and sent when the **uucico** daemon awakens.

Retrieving files from the public directory

In order to retrieve a file sent by **uuto**, you must use the **uupick** command. To execute **uupick**, enter a command like this:

uupick -s *system*

For example, *sara* would enter **uupick -s jane** and the **uupick** program searches the public directory for any files sent to you from *system*. If it finds any, it responds with the following prompt:

`from system` ***src_computer***: `file` *filename* `?`

src_computer is the name of the sender's computer and *filename* is the name of the file transferred. In the example above, if the **uuto** transfer to *sara* on *kate* is successful, *sara* sees the following **uupick** prompt:

`from system jane: file mytextfile ?`

Several options are available for responding to the **uupick** prompt. Two of the most useful are **m** *dir* and **d**. The **m** *dir* option tells **uupick** to move the file to directory *dir*. Once in *dir*, you can manipulate the file as you would any other file on your system. In the example above, *sara* enters the following response to the **uupick** prompt:

m $HOME

This causes *mytextfile* to be moved from the public directory to *sara*'s home directory. If no directory is specified after **m**, the file is moved to the recipient's current directory. The file can then be deleted from the public directory by entering **d** at the **uupick** prompt. You can quit **uupick** by entering **q**. Other **uupick** options are available; refer to **uupick**(C) for a complete list.

Executing commands on remote UNIX systems

Use the **uux**(C) command to execute commands on remote UUCP systems. For security reasons, the commands available for remote execution on a computer are often very limited. A computer's */usr/lib/uucp/Permissions* file lists the commands that can be executed remotely on that computer. If you attempt to execute a command not listed in this file, you receive mail indicating that the command cannot be executed on the computer in question.

The format of **uux** is:

uux [*options*] *command_line*

The *command_line* argument looks like any other UNIX command line, with the exception that commands and filenames might be prefixed with **sitename!**.

The following is an example of how to execute a command on a remote system. The command causes */tmp/printfile* on *rachel* to be sent to *rachel*'s default printer:

uux rachel!lp rachel!/tmp/printfile

The following is an example of how to execute a command on a local system, on files gathered by **uux** from remote systems. Suppose that your local computer is connected to both *rachel* and *kate*. Suppose also that you want to compare the contents of */tmp/chpt1* on *rachel* with */tmp/chpt1* on *kate*. To do so, enter the following command:

uux "diff rachel!/tmp/chpt1 kate!/tmp/chpt1 > diff.file"

This command compares the contents of the files on *rachel* and *kate* and places the output in *diff.file* in the current directory on the local computer. Since there is no site name prefixed to the **diff** command, the command is executed locally.

Note that, in the example above, the **uux** command line is placed in quotation marks. This is because it contains the redirect symbol (>). In general, place the **uux** command line in quotation marks whenever the command line contains special shell characters such as " < ", " > ", and " | ".

Dialing up remote systems

The **ct**(C) command connects your system to a remote terminal with a modem attached. The **cu**(C) command connects your system to a remote system. The remote system can be attached via telephone lines or via a simple serial line. These commands differ from the UUCP commands discussed earlier in that your session with the remote system is interactive. The remote system sees you as just another user on the system.

Connecting to a remote terminal

The **ct** command connects a local computer to a remote terminal equipped with a modem and allows a user on that terminal to log in to the computer. To do this, the command dials the telephone number of the remote modem. The remote modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a **getty** (login) process for the remote terminal and allows a user on the terminal to log in on the computer.

This command is especially useful when issued from the opposite end, that is, from the remote terminal itself. If you are using a remote terminal and you want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. To do so, simply call the computer, log in, and issue the **ct** command. The computer hangs up the line and calls your terminal back.

If **ct** cannot find an available dialer, it tells you that all dialers are busy and asks if it should wait until one becomes available. If you answer yes, it asks how long (in minutes) it should wait. If you answer no, **ct** quits.

The format of **ct** is **ct** [*options*] *telno* where *telno* is the telephone number of the remote terminal.

As an example, suppose that you have a terminal with a modem attached at home and that you want to log in to the computer at work from this terminal. To avoid long distance charges, first call your work computer and log in. Then issue the **ct** command to make the computer hang up and call your terminal back. If your telephone number is 555-1212, the **ct** command is:

ct -s 1200 5551212

The **-s** (speed) option tells **ct** to call the modem at 1200 baud. If no dialer device is available on the computer at work, you see the following message after executing **ct**:

```
The one 1200 baud dialer is busy
Do you want to wait for dialer? (y for yes):
```

If you type **n** (no), the **ct** command exits. If you type **y** (yes), **ct** prompts you to specify how long **ct** should wait:

```
Time, in minutes?
```

If a dialer is available when you enter the **ct** command, you see the following message:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. You are then asked if you want the line connecting your remote terminal to the computer to be dropped:

```
Proceed to hang-up? (y to hang-up, otherwise exit):
```

Since you want to avoid long-distance charges by having the computer call you, answer **y** (yes). You are then logged off and **ct** calls your remote terminal back.

As another example, suppose that you are logged in on a computer through a local terminal and you want to connect a remote terminal to the computer. The telephone number of the modem on the remote terminal is 5551000. To connect the terminal, enter the following command:

> **nohup ct -h -s 1200 5551000 &**

The **-h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer. After the command is executed, a login prompt is displayed on the remote terminal. The user can then log in and work on the computer just as on a local terminal.

Using two computers at the same time

The **cu** (call up) command makes your local computer call a remote computer and allows you to be logged in on both systems simultaneously. The remote computer does not have to be a UNIX system.

If the remote computer is a UNIX system, **cu** allows you to move back and forth between the two computers, transferring files and executing commands on both. Note that **cu** only allows you to transfer text files. You cannot transfer binary files with **cu**. To transfer binary files to a remote UNIX system, use **uucp**.

The format of the **cu** command is:

> **cu** [*options*] *target*

The *target* argument can take one of four forms:

phone number This is the number of the remote computer to which you want to connect. You can embed equal signs, which represent secondary dial tones, and dashes, which represent four-second delays, in the telephone number. A sample telephone number might be 4085551212--100. This number contains an area code and number, two dashes for an eight second delay, and an extension.

system-name This is the name of a system that is listed in the */usr/lib/uucp/Systems* file. The **cu** command obtains the telephone number and the baud rate of *system-name* from this file. The **-s**, **-n**, and **-l** options should not be used with *system-name*. To see the list of computers in the *Systems* file, enter **uuname**.

-l *line* This is the device name of the serial line connected to the remote computer. It has the form *ttyXX*, where *XX* is the number of a serial line.

-l *line dir* This connects directly with the serial line instead of making a telephone connection.

Once the connection is made, if the remote computer is a UNIX system, you are presented with a login prompt. Log in as you would if you were connected locally. When you finish working on the remote computer, log out as you would if you were connected locally, then terminate the **cu** connection by entering a tilde followed by a period (~.). (The tilde, when entered at the beginning of a line, is an escape character that tells **cu** to process the next piece of text itself, instead of sending it to the remote computer. The period is the **cu** command to terminate the session. Other commands are available.)

As an example, suppose that you want to log in to a remote UNIX computer via the telephone lines. Suppose also that the remote computer's number is 555-1212. To connect to the remote computer, enter the following command:

    cu -s1200 5551212

The **-s1200** option causes **cu** to use a 1200 baud dialer. If the **-s** option is not specified, **cu** uses the first available dialer at the speed specified in the */usr/lib/uucp/Devices* file.

When the remote UNIX system answers the call, **cu** notifies you that the connection has been made by displaying the following message:

    Connected

Next, you are prompted for your login:

    login:

Enter your login and password. Once you enter this information, you can use this computer as if you were logged in locally. When you are finished, log out and then enter:

~.

This terminates the **cu** session.

Transferring text files with take and put

Several command strings are available with **cu** that allow your local computer to communicate with a remote UNIX system. Two of the most useful are **take** and **put**.

The **take** command allows you to take files from the remote computer to the local computer. Suppose, for example, that you want to copy a file named *proposal* in the current directory of the remote computer to the */tmp* directory on the local computer. To do so, enter the following command:

~%**take proposal /tmp/proposal**

Note that you have to prefix a tilde and a percent sign (~%) to the **take** command, and that the tilde must be placed at the start of a line. For this reason, it is a good idea to press ⟨Enter⟩ before using **take**.

The **put** command does the opposite of **take**. It puts files from the local computer onto the remote computer. Suppose, for example, that you want to copy a file named *minutes* from the */usr/spool/uucppublic* directory on the local computer to the */tmp* directory of the remote computer. Suppose also that you want the file to be called *minutes.9-18* on the remote computer. To do so, enter the following command:

~%**put /usr/spool/uucppublic/minutes /tmp/minutes.9-18**

As with the **take** command, you have to prefix a tilde and a percent sign (~%) to the **put** command, with the tilde coming at the beginning of a line.

put and **take** rely on other programs existing at either end of the connection; **put** needs **stty** and **cat**, while **take** needs **echo** and **cat**. They may not work if you use **cu** to connect to a computer that does not have these programs (for example, a computer running DOS). **put** and **take** can only copy text files; if you want to send a binary file, uuencode it before transmission. (See the *Mail and Messaging Guide* for an explanation of how to use **uuencode** and **uudecode**.)

The **cu** command cannot detect or correct transmission errors. After a file transfer, you can check for loss of data by running the **sum** command on both the file that was sent and the file that was received. This command reports the total number of bytes in each file. If the totals match, your transfer was probably successful. See the **sum**(C) manual page for details.

For example, here is a sample **cu** session:

```
# uname
jane
# cu rachel

login:andrew
Password:
TERM = (ansi)

# cd /
# lf
.hushlogin     autoexec.bat   date.dat      mnt/          tmp/
.lastlogin     autotest*      dev/          opt/          u/
.login         bin/           dos           sc*           unix
.mailrc        boot           etc/          sco_extra.ps  unix.old
.profile       command.com    hj*           server/       usr/
.rhosts        config.sys     install/      sfmt*         var/
.utillist2     core           lib/          shlib/        vmstat.dat
README         country.sys    lost+found/   tcb/
# compress README
# uuencode README.Z README.Z>readme.uue
# ~%take readme.uue
sh -c "stty -echo;test -r readme.uue&&(echo '~>':readme.u
ue;cat readme.uue;echo '~>');stty echo" ~>:readme.uue
1234567+
# ~!
# uname
jane
# ls message.file
message.file
# exit
# uname
rachel
# ~%put message.file
sh -c "stty -echo;(cat - >readme.uue)||cat - >/dev/null;st
ty echo"
12345678901234567+
# exit
~.
# uname
jane
# uudecode readme.uue
# uncompress README.Z
#
```

Chapter 9

Using a secure system

Every computer system needs protection from people accessing the computer, disks and system files without permission. The SCO OpenServer system provides built-in security features not present in other UNIX systems. These features apply to all users of the system and are maintained by the system administrator.

This chapter explains the following:

- how system security works (page 208)
- login security (page 208)
- what to do if you cannot log in (page 209)
- password security (page 209)
- changing passwords (page 210)
- file security (page 211)
- general security tips (page 212)
- using commands on a trusted system (page 213)
- data encryption (page 216)

This chapter describes security from the viewpoint of the ordinary user. If you find that your system does not use a feature discussed in this chapter, your administrator has switched it off in favor of standard non-trusted behavior. For full details of the security systems implemented on the SCO OpenServer system, refer to Chapter 5, "Maintaining system security" in the *System Administration Guide*.

If you become aware of suspicious activity on the system, you are advised to contact your system administrator immediately.

How system security works

System security is built on two foundations; being able to validate the identity of a user, and being able to determine whether a given user has permission to carry out a task. When you log in, the system uses your login to check the password file; when you type your password, the system encrypts it and compares it with the (encrypted) copy of your password that it already knows. This acts as a check on your identity. If you disclose your password to someone else, they can log in as you.

Access to the files on the system is controlled by your permissions; see "Access control for files and directories" (page 121). Note that the system administrator or *root* user can read or write any file they want to. Thus, the most important password on the system is the *root* password.

In addition to controlling file access on the basis of your login name, the system controls access to system services. Whenever you run a program, the process it gives rise to inherits your authorizations and privileges. Thus, if you lack the appropriate privilege, you may not be able to use the **ps** command to check on other user's processes, to use **chown** to change the ownership of files, or to use **su** to run programs under another login. Again, system authorizations are assigned on the basis of your login; the *root* account is allowed to do anything. (Authorizations are assigned on a per-subsystem basis, while privileges are assigned for kernel based operations.)

Login security

- When you enter your password correctly, the last times you successfully and (if applicable) unsuccessfully logged in are displayed:

    Last successful login for *login: date and time* on tty*xxx*
    Last unsuccessful login for *login: date and time* on tty*xxx*

    If these times do not match your actions, consult your administrator immediately. Someone might have tried to log into your account. (For example, if you come in to work on a Monday and the system indicates a failed login attempt on Saturday night, when you were at home, you should be suspicious.)

- Be careful how you type in your password.

- When you enter your password and the system reports an error, although you believe your entry to have been correct, tell your security administrator immediately. Check the reported last login time against the current time. If there is a discrepancy it is possible that a spoofing program has stolen your password. (A spoofing program is a hacking tool designed to collect passwords. It puts a display on a terminal that resembles a login prompt. When a user types their password and username into it, it sends the information to its owner, logs them in, then lies in wait for the next user.)

What to do if you cannot log in

If you cannot log in, go through the following list of possible reasons:

- The system administrator has given your password a lifetime, which has now expired. Ask the administrator to change your password and reopen your account.

- The system administrator has set a limit to the number of unsuccessful log-in attempts allowed for your account or your terminal. When you exceed this number your account or terminal is locked automatically. Ask the administrator to reopen the account or unlock the terminal. If you feel that you entered your login details correctly, tell the administrator immediately. It is possible that the system has been interfered with.

- The system administrator has locked your account or terminal. To continue work, you must ask the administrator to reopen the account.

- The system administrator has set a date by which your password expires. When your password expires, you are prompted to change it.

- You have forgotten your password. Ask the system administrator to change it.

Password security

It is your responsibility to protect your password. The careless use and maintenance of passwords represents the greatest threat to the security of a computer system. The security administrator can configure the system to be as restrictive or open as desired. Password restrictions such as length, complexity, and lifetime may be enforced by the system.

Here are some basic guidelines for choosing and maintaining passwords:

- A password should be at least eight characters in length and include letters, digits, and punctuation marks. For example: **frAiJ6***.

- Do not use a password that is easy to guess. A password must not be a name, nickname, proper noun, or word found in the dictionary. Do not use your birthday or a number in your address.

- Do not use words spelled backwards.

- Do not start or end a password with a single number. For example, do not use **terry9** as your password.

- Use different passwords on different machines. Do not make the passwords reflect the names of the machines.

- Always keep your password secret. A password should never be written down, shared with another person, sent over electronic mail, or spoken. Treat your password like the PIN number for your instant teller card.

- Never reuse a password. This increases the probability of someone guessing it.

- Never type a password while someone is watching your fingers.

Changing your password

The security administrator decides whether you can change your password for yourself. The administrator can also set a minimum time period between changes of password.

If you are not allowed to change your password

If you are not allowed to change password, and you try to use the **passwd**(C) command, the following message appears:

```
Password cannot be changed.  Reason: Not allowed to
execute password for the given user.
```

In this case, you must ask the administrator to change your password.

If you are allowed to change your password

If you are allowed to change your password, the administrator sets up your account to allow you to specify the password of your choice or to have the system generate one for you.

When you use the **passwd**(C) command, you are prompted for your current password:

```
Old password:
```

When you type it in correctly, the date and time of your last change of password are displayed:

```
Last successful password change for login: date and time
Last unsuccessful password change for login: date and time
```

Make sure that these messages reflect your last attempts to change password. If they do not, tell your administrator immediately.

Follow the instructions on the screen to pick your own password or have the computer generate one for you.

File security

Follow the guidelines below when you are creating, copying, and moving files. The list also includes security tips related to your startup scripts.

- When you create a file or directory, your **umask** determines the permissions given to the file or directory. For information about **umask**(C) see "Setting the default permissions for a new file" (page 124). Newly created files and directories should only be accessible by you (the owner) or the group. If you wish to share files with other users, change the permissions on those files individually.

- When you use **cp**(C) to copy an SUID file owned by someone else, the SUID bit is reset. (This is a security precaution.) Note that when you execute a SUID file, it has access to all your files and directories.

- When you use **cp** to copy a file so as to create a new file, the new file takes the permissions of the original file. Remember to check the permissions of the new file and, if necessary, change them using the **chmod**(C) command.

- Remember that temporary directories are world-readable.

- Use **ls**(C) to check the permissions on your shell, mailer, startup files, and home directory. If the files can be read and modified by other users, change the permissions using **chmod** so that only you have access to them. If the directory can be executed by other users, those users can **cd** to it; if the directory can be written to by other users, they can remove files within it. Change the permissions on your home directory so that only you have write or execute permissions on it.

- Make certain that sensitive files are not publicly readable.

6

Security for files in sticky directories

A directory with the sticky bit set means that only the file owner and the superuser can remove files from that directory. Other users are denied the right to remove files, irrespective of directory permissions. Only the superuser can place the sticky bit on a directory.

A sticky directory contains a "t" at the end of the permissions field, as in this example:

```
drwxrwxrwt   2 bin      bin      1088 Mar 18 21:10 tmp
```

Other security tips

- Log out before leaving a terminal.
- Use the **lock**(C) utility when you leave your terminal, even for a short time. The **lock** command requests a password at the time of use, and then locks the terminal until the password is re-entered.

  Note that **lock** is ineffective if you are using a shell which supports job control (such as the C shell or the Korn shell), the **shl** layers system, any pseudo-tty system that permits you to switch between virtual terminals, or an X-windows terminal. If any of these conditions apply, you should log out instead.
- Keep disks or tapes containing confidential data (program source, database backups) under lock and key.
- If you notice strange files in your directories, or find other evidence that your account has been tampered with, tell your system administrator.

Using su to access another account

The **su**(C) command allows you to become another user without logging off. **su** cannot be used to simply assume the login of another user; instead, **su** can be used under four circumstances:

- The superuser (*root*) can "su" to any account.

- A user with the **su** authorization can "su" to *root* (for example, to become *root* temporarily after having su'd to a different account).

- Users can "su" to their own accounts.

- A system daemon can "su" to an account.

- Under "traditional" security, any user can "su" to any account with that account's password.

To use **su**, the appropriate password must be supplied (unless you are already *root*). If the password is correct, **su** executes a new shell with the real and effective user ID's set to that of the specified user. (If the system is running under tight security, the login user ID is unchanged; otherwise that, too, is changed to the ID of the specified user.)

Using commands on a trusted system

The use of commands is restricted on a trusted system. You can issue certain commands only if the security administrator has given you the appropriate "authorization". This section describes the different types of authorization and how they affect your use of commands. (Authorizations are important if your system is working at the Improved or High security levels; at low or traditional security levels most authorizations are available to users by default.)

To determine the security level your system is working at, type:

    /tcb/bin/secdefs

This command reports the nearest matching security level. (There are a number of security systems that your administrator can customize. Consequently, your system may fall between two levels in some respects.) For more information, see **secdefs**(ADM).

Authorizations

The security mechanism has two types of authorization: kernel and subsystem. A kernel privilege allows you to run specific processes on the operating system. A subsystem authorization allows you to use the commands of a specific protected subsystem.

The kernel privileges are as follows:

execsuid
: allows you to run SUID (set user ID) programs. An SUID program gains access to all the files, processes, and resources belonging to the person running the program and the owner of the program file.

chmodsugid
: allows you to change the setuid and setgid attributes of a file or directory, using the **chmod**(C) command. Without this permission you cannot create SUID files, which grant the permissions of the owner of the file to whoever executes them, as described in "Access control for files and directories" (page 121).

chown
: allows you to change the ownership of files using the **chown**(C) command.

Other kernel privileges include **suspendaudit, configaudit**, and **writeaudit**.

There are two levels of subsystem authorization: primary and secondary. Primary authorizations are given to administrators and are fully described in the *System Administration Guide*. However, they can be given to ordinary users as well. Some primary authorizations are:

mem
: allows you to use **ps**(C) to check the status of other users' processes, and **ipcs**(ADM) to report the status of interprocess communication. Without this authorization, you can only use these commands to report on processes belonging to you.

terminal
: allows you to use **write**(C) to communicate with other users. If you use **write** without the authorization, any control codes and escape sequences in your message are converted to printable characters.

Other primary authorizations include **audit, auth, backup, cron, lp, sysadmin**, and **root**. (See **authorize**(F) for information on these authorizations.)

A secondary subsystem authorization allows you to use a subset of the commands of a subsystem as an ordinary user (that is, you are not given administrative privilege). Secondary authorizations are described below:

audittrail
: allows the use of the audit subsystem to monitor your own activities only. This can be useful for debugging of programs because a detailed record of system calls is generated by the audit daemon. For more information, see "Using the audit subsystem" in the *System Administration Guide*.

printqueue
: allows you to view other users' jobs on the print queue.

printerstat   allows you to use **enable**(C) and **disable**(C) to change the status of printers.

queryspace   allows you to use **df**(C) to query the amount of space available on the filesystems.

su   allows you to use **su**(C) to access another account (including *root*). Without this authorization, users can only access their own accounts.

Other secondary authorizations include **passwd, create_backup, restore_backup,** and **shutdown**.

Listing authorizations and running authorized commands

The **auths**(C) command allows you to list your kernel privileges, and to start up a shell so that you can issue commands with specific authorizations.

- The **auths** command without arguments lists your kernel privileges. For example:

```
$ auths
kernel privileges: execsuid,chown
```

- The **auths** command with the -a option allows you to specify a restricted set of one or more of your authorizations. For example, the user with *execsuid* and *chown* authorizations can restrict their use to the chown authorization:

```
$ auths -a chown
$ auths
kernel privileges: chown
```

To restore your authorizations, leave the shell started by the **auths -a** command.

- The **auths** command with the -r option allows you to specify which of your authorizations you wish to remove. For example:

```
$ auths -r chown
$ auths
kernel privileges: execsuid
```

Leave the shell started by the **auths -r** command to restore your authorizations.

- The **auths** command with the -c option allows you to issue a command instead of starting an interactive subshell. In the example below, chown authorization is removed and then the **auths** command is issued. The result is a line listing the user's authorizations; the chown authorization is not included.

```
$ auths -r chown -c auths
kernel privileges: execsuid
```

When the user executes another list, the chown authorization is restored:

```
$ auths
kernel privileges: execsuid,chown
```

Data encryption

If you have sensitive data that requires greater protection than that provided by access permission, you can encrypt the data. The encrypted file cannot be read without a password. If somebody tries to read the encrypted file without a password, it cannot be understood.

You will only have data encryption capabilities if the **crypt**(C) software is installed on your system. This software is available only within the United States and must be requested from your distributor.

A brief summary of encryption commands appears in the following table:

Command	Description
crypt	Encode and decode files. Reads from the standard input or keyboard and writes to the standard output or terminal.
makekey	Generates an encryption key.
ed -x	Edits an encrypted file, or creates a new encrypted file using the **ed** editor.
vi -x	Edits an encrypted file, or creates a new encrypted file using the **vi** editor.
ex -x	Edits an encrypted file, or creates a new encrypted file using the **ex** editor.
edit -x	Edits an encrypted file, or creates a new encrypted file using the **edit** editor.
X	Encrypts a file while in the editor mode (**ed**, **ex**, or **edit**).

crypt—encode/decode files

The **crypt**(C) command encodes and decodes files for security. When using **crypt**, you have to assign a password (key) to encode the file. The same password is used to decode the file.

If you do not give a password with the **crypt** command, the system prompts you for one. For security, the screen does not display the password as you type it in.

Password security is the most vulnerable part of the **crypt** command. The best way to ensure your security is to select an uncommon group of characters. The password should be no more than eight letters or numbers long.

A file can be encrypted in the shell mode using **crypt**, or in the edit mode using the **-x** or **X** option. When you are ready to decrypt the file, you can use the **crypt** command in the shell mode. The following is the command format to encrypt a file:

    crypt < *oldfile* > *newfile*

The system prompts you for a password.

Before removing the unencrypted *oldfile*, make sure the encrypted *newfile* can be decrypted using the appropriate password.

To decrypt a file, redirect the encrypted file to a new file you can read. The command to decrypt a file is as follows:

    crypt < *encrypted_file* > *new_filename*

Shell Programming

Chapter 10

Configuring and working with the shells

This chapter describes the different shells you may be working in. It explains the special features they provide to make your work easier, how to use variables to store information used by programs, and how to use aliases to define new commands recognized by the shells. It also describes how the shells process your instructions.

This information is contained in the following topics:

What is a shell?

Shells are interactive programs that provide a "glue" that fastens other programs together. They perform the following tasks:

- Execute the commands typed at the shell prompt.
- Find and execute other programs on command.
- Interpret the wildcard characters in filename specifications.
- Interpret complex regular expressions.
- Permit redirection of input and output.

- Construct pipelines containing several programs operating in sequence on the same data stream.

- Process shell scripts (a collection of shell commands that can control execution of other programs).

- Perform some basic tests on data and files.

The shells provide a command line interpreter that responds to typed commands, and a programming language that allows you to create scripts.

Your work takes the form of a dialogue with your shell, which acts as the interface between you and the rest of the system. Like all good interfaces, your shell allows you to customize it to make life easier.

What the different shells are for

Three different command oriented shells are available for the SCO OpenServer system. You can choose to work with any one of them. The shells are as follows:

The shells

Name	Filename	Features
Bourne Shell	*/bin/sh*	- First shell to be developed. - Wildcards, basic command language. - Available on the SCO OpenServer system.
C Shell	*/bin/csh*	- Different language syntax from Bourne and Korn shell family (similar to the C programming language). - Command history recall (permits reuse of recently issued commands without retyping them). - Aliases (the ability to define alternative names for commands). Limited ability to redirect input and output.

The shells
(Continued)

Name	Filename	Features
Korn Shell	*/bin/ksh*	• Compatible superset of Bourne shell facilities.
		• Command history editing (edit and reissue previously typed commands interactively).
		• Aliases (the ability to define alternative names for commands).
		• Job control (the ability to run processes in the background and manipulate background processes).
		• Extended language syntax (permits more complex scripts to be written).
		• Recommended as the shell of first choice.

The SCO shell is a different type of shell: a menu-driven interface that cannot execute scripts directly. It is discussed in Chapter 1, "Using SCO Shell" (page 11).

In this chapter and the next we will be concentrating on the Korn shell: specifically, on those features of the Korn shell that are also available to the Bourne shell. Where additional Korn shell facilities are introduced, they are explicitly identified as such because they are not available under the Bourne shell.

Note that we do not recommend the C shell to new users. C shell syntax is nonstandard, and there are a number of features present in the Bourne and Korn shells that are not present in the C shell.

Identifying your login shell

Because the different shells understand different commands, it is important to know which shell you are working in. To find out what your login shell is, type **grep $LOGNAME /etc/passwd** (**LOGNAME** is the environment variable that stores your login name). You will see something like the following line:

```
charles:*:13079:1014:Charles Stross:/u/charles:/bin/ksh
```

The last field of this line (after the last colon (:)) is the login shell executed for the user named in the first field on the line.

(This line is a record from the */etc/passwd* file, a database that identifies the home directory, login name, group ID, permissions, and login shell for every user on the system. Only the system administrator can change this file.)

You can run a shell interactively as a subprocess (often called a subshell) by typing its name (for example, **csh**). Your subsequent commands are interpreted by the subshell until you type the command **exit** to quit the shell. The system is set up to load one particular shell for you every time you log in.

If you want to change your login shell, for example to switch to the Korn shell on a permanent basis, ask your system administrator.

What happens when you log in

When you log in, the system first asks for your user name (to identify your home directory and permissions), then your password (to confirm your identity). Having identified your account, the system then starts a shell for you.

If you are using the Bourne shell (**sh**) or the Korn shell (**ksh**), the shell first executes the commands stored in the generic environment file */etc/profile*, then the commands stored in the personalized environment file called *.profile* located in your home directory, if that file exists. A Korn shell additionally looks for a file called *.kshrc*; if it exists, this is executed after *.profile*.
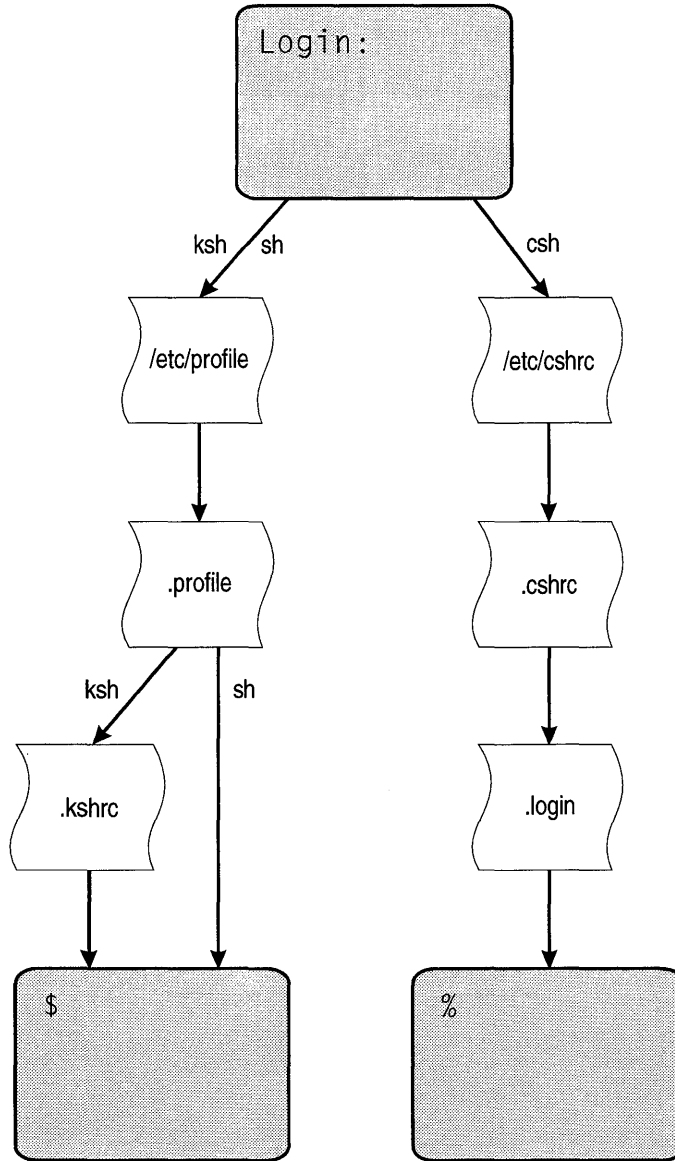
If you are using the C shell (**csh**), the shell executes the commands stored in the file */etc/cshrc*, then any commands present in a file called *.cshrc* in your home directory, if that file exists. The shell then looks for a file called *.login*; if it exists, any commands in it are executed. Note that the default prompt for the Korn and Bourne shells is the "$"; the C shell's default prompt is the "%".

You can find annotated examples of *.profile*, *.kshrc*, *.login*, and *.cshrc* files in Appendix D, "Sample shell startup files" (page 419).

These files set up your work environment. They contain commands to configure your terminal type and to set up various *environment variables* (see "Understanding variables" (this page) for details). The login files also contain any other commands that you want to have executed every time you log in.

The login procedure displays a lot of information that you may not need or want. If you specifically do not want to see system messages (such as your last login date, the message of the day, or the system copyright message), create an empty file in your home directory called *.hushlogin*. For example:

```
$ touch .hushlogin
$
```

If you execute this command, you must use the **-a** option to **ls** in order to list the newly created file: **ls** on its own does not list "dot" files. The **touch** command updates the last access time of a file; if you give it a filename which does not exist, it creates an empty file of that name.

Note that the ability to use *.hushlogin* may be disabled if your system is running at an enhanced security level. If this is the case, you will see the login messages whether there is a *.hushlogin* file in your home directory or not.

Understanding variables

The shells provide facilities for storing useful information and transferring it between programs. Among these is the ability to handle variables (named pieces of text or numbers, that can be used in a variety of ways).

Variables have many uses. For example, if you frequently need to **cd** to */u/work/systems/Admin*, you could define the variable **ADMIN** to be */u/work/systems/Admin*, then type **cd $ADMIN** to change to that directory.

Variables consist of a name (or label) and an associated value. In the example above, the variable is named **ADMIN**; its value is */u/work/systems/Admin*. You refer to the value of a variable by prefixing its name with a " $ " symbol. When the shell reads the " $ " symbol it checks the subsequent text to see if it is a variable name (such as **NAME**), and replaces the input text $NAME with the value of **NAME**.

10

There are two types of variable available to you:

Shell variables
> These are created within a shell and are used to temporarily store information and to control the execution of shell scripts (see Chapter 11, "Automating frequent tasks" (page 245)). Shell variables are not visible to any other program, and are lost when the shell terminates.

Environment variables
> All programs running on the system have a special memory area called an environment. When a program is run, it inherits a copy of its parent program's environment, complete with any variables stored in it. Environment variables are used to pass configuration information to child processes executed by the shell. They are created by *exporting* a shell variable into the shell's environment, which makes them visible to all programs subsequently executed under that shell (see "Exporting variables to the environment" (page 230)). However, it is not possible for a child to alter its parent shell's environment.

The sections below explain how to create and refer to shell variables and environment variables.

Setting shell variables

To set a variable in the shell, use an equal sign to assign it a value. If the variable does not already exist, it is created. For example:

> **MYVARIABLE=hello**

It is common practice to use all uppercase letters for the name of variables, to distinguish them from UNIX system commands (that are almost always lowercase).

To refer to the value of a variable, prefix the variable's name with a " $ " symbol. If you omit the " $ ", the shell will assume you are referring to the name of the variable, not its current value. For example, in the Bourne and Korn shells:

```
$ MYVARIABLE=hello
$ echo MYVARIABLE
MYVARIABLE
$ echo $MYVARIABLE
hello
$
```

The C shell equivalent of this is as follows:

```
% set MY=hello
% echo MY
MY
% echo $MY
hello
%
```

You may sometimes see variable names enclosed in braces ({}) within a reference. The braces are used to delimit the name of the variable. For example **echo ${MYVARIABLE}** could be used instead of **echo $MYVARIABLE.** This is particularly useful when you want to concatenate the contents of a variable with another word. For example:

```
$ MYVARIABLE=hello
$ echo $MYVARIABLE
hello
$ echo ${MYVARIABLE}_there
hello_there
$ echo $MYVARIABLE_there

$
```

In the third **echo** command, because **MYVARIABLE** is not separated from "_there" the shell tries to substitute a variable called **MYVARIABLE_there** (which does not exist).

It is a good idea to make a habit of placing variable names in parentheses whenever there is any doubt, to reduce the likelihood of this kind of error.

Setting environment variables

The shells use some variables to configure their operations. For example, the Bourne shell and Korn shell provide a facility to make the shell notify you when mail arrives. To use it, set the shell variable **MAIL** to the name of the file in which you keep your mail (usually *.mailbox*). If the file grows, **ksh** will notify you. The existence of the **MAIL** variable is used by the shell as a flag to indicate that it should notify you whenever new mail arrives. You can set the variable within the shell, or set it in one of the profile files executed at login; the presence or absence of the variable affects the way the shell behaves.

The following is a list of the environment variables automatically set by the Korn shell (see **ksh**(C)). The other supported shells have a similar list of variables; for details see **sh**(C) and **csh**(C).

ERRNO   Set the value of the last error condition returned by a failed system call.

LINENO   Set to the current line number of the script or function being executed.

OLDPWD	The previous directory set by **cd**(C).
OPTARG	The value of the last option argument processed by the **getopts**(C) special command.
OPTIND	The index of the last option argument processed by the **getopts** special command.
PPID	The process number of the parent of the shell.
PWD	Present working directory.
RANDOM	A random integer number (in the range 0 to 32767).
REPLY	Set by the **select** statement (see "Generating a simple menu: the select statement" (page 283)) and by the **read** special command (see **ksh**(C)) when no arguments are supplied.
SECONDS	The number of seconds since **ksh** was invoked.

The following environment variables are also used by the Korn shell:

CDPATH	The search path for the **cd** command.
COLUMNS	The width of the edit window for the shell edit modes and for printing select lists.
FCEDIT	The default editor name for the **fc**(C) command.
FPATH	The search path for function definitions.
IFS	Defines the character used as the internal field separator.
HISTFILE	The pathname of the file that will be used to store the command history.
HISTSIZE	The number of previously entered commands that are accessible by the shell; the default is 128.
HOME	The default home directory for the **cd** command.
LINES	The number of lines on the terminal. Used by **ksh** and some other programs when presenting menus; the default is 24.
MAIL	A mailfolder. If it grows, the shell notifies you that mail has arrived.
MAILCHECK	The time interval in seconds between checks for new mail.
MAILPATH	Tells the shell to inform the user of any modifications to the specified files that have occurred within the last **MAILCHECK** seconds.

PATH	The search path for commands.
PS1 ... PS4	Prompt strings (see **ksh**(C) for more details).
SHELL	The pathname of the shell.
TERM	The terminal type; used by many programs that write to the screen.
TMOUT	The number of seconds of inactivity after which the shell will automatically terminate; a value of 0 means that the shell will not automatically terminate.

Many programs other than the shells look for specific variables every time they run; such variables are used to control the execution of these programs. For example, **vi** checks for a variable called **EXINIT** whenever it starts up. If any **vi** options are specified in **EXINIT**, **vi** sets them accordingly. Likewise, **mail** checks for a variable called **MAILRC** which specifies the startup file from which **mail** reads its options. By setting some environment variables, usually at login time, you can customize these programs to your requirements.

Exporting variables to the environment

Variables stored in the environment are visible to you within the shell; but variables you set within your shell session are not accessible to other programs running in the environment until you make them so by explicitly exporting them.

To export a shell variable to the environment, use the **export** command. For example, in the Bourne and Korn shells:

```
$ FOO=bar
$ export FOO
```

In the Korn shell, the following alternative form exists:

```
$ export FOO=bar
```

In the C shell:

```
% setenv FOO bar
```

This will cause the variable **FOO** to be exported to any processes started from within the current shell.

You can export more than one variable at a time with the **export** command by listing a set of variable names to be exported. (There may already be an **export** command in your startup file. In this case, add the name of the additional variable to the end of the list of variables for export.) For example:

```
    .
    .
    .
#
PATH=/bin:/usr/bin:/u/bin:/usr/local/bin:/local/bin:${HOME}/bin:
LOGNAME=charles
MAIL=${HOME}/.mailbox
    .
    .
    .
export PATH LOGNAME ...
    .
    .
    .
```

A sample login script

Here is a simplified *.profile* file:

```
# -- aliases; Korn shell only
alias dir='ls -al'
alias del='rm'
alias mail='op email'
# -- environment; Bourne and Korn shells
PATH=/bin:/usr/bin:/usr/local/bin:$HOME/bin:
set -o emacs # Korn shell history editing
MAIL=/u/charlie/.mailbox
PS1='$PWD>'
PS2='$PWD>>'
TERM=wy60
export PATH MAIL PS1 PS2 TERM
echo # prints a blank line
```

This file is run automatically when you log in, if your login shell is the Bourne or Korn shell. The first line of this file begins with a " # " character; this introduces a "comment", a line of text which the shell ignores. Comments are used to make shell scripts easier to understand.

The first section of this file contains a list of aliases. Aliases define synonyms for commands. For example, if you are used to the DOS environment, you will be familiar with the command **dir** to obtain a directory listing. To set up **dir** as an alias for the corresponding command (**ls**), the following line is executed:

alias dir='ls -al'

Once this command has been executed, every time you type **dir** the login shell will replace "dir" with "ls -al". Note that this form of alias syntax is recognized by the Korn shell: the C shell uses a different syntax, and the Bourne shell does not provide aliases. (A Bourne shell startup file will therefore omit these lines.) See "How aliases are executed" (page 238) for a detailed explanation of aliases.

After alias expansion, a number of environment variables are set. These are then exported with the **export** command, so that they are available to subprocesses running under the current login shell. In general, variables set in the startup files are important to the smooth running of your login session. For example, note the reference to the variable **PATH**. This contains a list of directories, separated by colons.

When you type a command without specifying any directory, the shell looks for a file of that name among the directories in **PATH** (unless the command is built into the shell, in which case it is executed without a search). If it finds a file of that name it tries to execute it; otherwise, the command fails because the shell could not find the correct program to run. If you remove the **PATH** variable you will be unable to execute programs without specifying their full pathname. For example, you would have to type **/bin/vi** instead of just **vi** to run the editor.

Resetting the environment

From time to time you may want to reset your environment; either because you have changed your *.profile* or *.login* files, or because you've erased a variable.

To reset the environment, re-execute your login file. Under the Bourne or Korn shells, type the following:

```
$ . $HOME/.profile
```

Under the Korn shell (but not the Bourne shell) you can also type the following:

```
$ . ~/.profile
```

(The dot is a command to execute the following file.)

Under the C shell, type the following:

```
% source $HOME/.login
```

Some features to make life easier

This section contains some ways of working with the shell. You can do any of the following:

- Use environment variables or aliases to make your Korn or Bourne shell prompt tell you what your current directory is, or to display other useful information.

- Use the Bourne or Korn shell **trap** command to specify actions to take when the shell receives a signal; for example, to execute a logout script.

- Use the Korn shell or C shell to recall and edit previously issued commands.

- Define complex aliases or functions to shorten long command lines or redefine commands.

Making your prompt tell you where you are

One variable in *.login* or *.profile* that you might want to adapt is your prompt variable. The prompt variable contains the character or characters that the shell prompts you to enter a command with. The main prompt string is contained in the Korn shell or Bourne shell variable **PS1**. If you press ⟨Enter⟩ to start a new line without completing a command, the shell will prompt you with the **PS2** (second level) prompt variable. (If you are using the C shell, your prompt string is stored in the variable **prompt**.)

For example:

```
$ echo "
> hello, world!
> This is a test.
> "

hello, world!
This is a test.
$
```

In the Korn shell only, you can make the prompt string display your current working directory by editing *.profile* and setting the variable (as in the *.profile* above) like this:

PS1='$PWD'

You must then reset your environment for the change to take effect. See "Resetting the environment" (page 232) for details.

Note that the variable **PS1** is almost certainly present in one of your startup files already; if you insert the example line above and forget to remove the old setting, then whichever version of **PS1** was specified last in the file will be used.

Adding a logout script

A logout script is a short list of commands that are typically executed when you log out; for example, to issue the **clear** command to clear your screen, and record the amount of time you spent working.

To create a logout script, edit a file called *.logout*, enter commands into it, then make it executable with **chmod +x** *.logout*. To have it execute automatically when you log out, add the following line to your *.profile* file (Bourne or Korn shells):

```
trap '$HOME/.logout' 0
```

A typical *.logout* script might look like the following:

```
clear
banner $LOGNAME "is out"
```

When you log out by pressing ⟨Ctrl⟩D or typing **exit**, you are sending a *signal* to the shell. A signal notifies the shell that a special event has occurred, and the shell should take action. Several different types of signal are available to the system, but the one you send by logging out is signal 0, called EXIT. (Actually, the ⟨Ctrl⟩D sends a pseudosignal, but this distinction will be dealt with later.)

See "Using signals under the UNIX system" (page 166) for a detailed explanation of signals and how to use them.

The notation $HOME/*.logout* is interpreted by the shell; it looks in the variable **HOME** and substitutes its contents. $HOME is your home directory pathname, so this enables the shell to execute your logout script wherever you are.

Note that if your session is being run in a windows environment, logging out will kill the window as well as the session; in such a case, a logout script may be of no use.

Recalling and editing previous commands

The Korn and C shells provide facilities for making it easier to enter commands:

C shell   Provides command history recall. The shell maintains a list of previously executed commands; you can refer back to entire command lines or sections of commands, and re-execute them if necessary.

Korn shell   Provides command history editing. In addition to remembering a list of previously executed commands, the Korn shell allows you to edit previous command lines interactively, using the keystrokes of the **vi** or **emacs** text editors.

Many people prefer the facilities offered by the Korn shell to those of the C shell because of the interactive editing feature.

Korn shell history editing

Every time you issue a command to the Korn shell, in addition to executing the command, the shell adds it to a list of previously executed commands.

To view the list of previously executed commands, issue the **history** command. This displays a number (up to the number set in **$HISTSIZE**) of previously issued commands. **$HISTSIZE** is set in *.profile.*

Before you can recall and edit the history list directly, you must issue the **ksh** command **set -o vi** to set the **vi** editing mode in the shell. Similarly, the following command line sets **emacs** editing mode in the shell:

> **set -o emacs**

Either of these commands may be issued automatically by your *.profile* file. **set -o** on its own displays all the current **ksh** options; you may want to see what options are available.

To enable history editing permanently, add the following to your *.profile* file:

> **if [-z "$VISUAL" -a -z "$EDITOR"]**
> **then**
> > **set -o vi**
> **fi**

(This may also be found in your *.kshrc* file.) See **ksh**(C) for details of the **VISUAL** and **EDITOR** variables; see **test**(C) for an explanation of the [...] notation and the options used in the example.

Once the **vi** option, for example, is available, you can edit your command history using the **vi** editing keys. The shell initially behaves like **vi** in text insertion mode; if you type a command it inserts text, and when you press ⟨Enter⟩ it executes the line you just typed. Here is a quick overview:

- To switch into **vi** command mode, press ⟨Esc⟩. You can use the **vi** cursor movement keys "k" (up, towards older commands), or "j" (down, towards more recent commands). You can also move left or right along the line using the "h" (for left) or "l" (for right) keys.

- Previously issued commands are displayed on the current line as if you are editing them in **vi** in a window one line high; each time you type ⟨Enter⟩ the cursor is effectively repositioned to the bottom of the file. New commands are added to the end of the file as you type.

- When you press ⟨Enter⟩, the currently visible command line is executed by the shell, whichever mode you are in. To repeat the previous command, type **r**.

- Many other **vi** editing keystrokes also work in history editing mode. You can delete words with the **dw** command, switch to text insertion mode with the **i** command, and search for a command containing a piece of text using the */string* command (where *string* is the text to search for).

For example, if you want to reissue the last command you typed, simply press ⟨Esc⟩ (to switch to command mode), then "k" to move up to the previous command, then ⟨Enter⟩ to execute the command.

A full list of the available **vi** mode command editing keystrokes is given in the **ksh**(C) manual page. For an introduction to the **vi** editing keystrokes, see "A quick tour of vi" (page 132).

Note that the contents of the history file are maintained across logouts and environment resettings. Accordingly, previously executed command lines are still available for editing or re-execution even after something like the following:

> . $HOME/.profile

C shell history editing

The C shell does not let you edit previous commands interactively. However, you can recall entire command lines or portions of commands, and make changes to them.

To repeat the last command you typed, use the following command:

> % !!

The exclamation mark (!) tells **csh** to expect a history command, and the second exclamation mark specifically refers to the last command entered.

To display your command history, use the following command:

% `history`

The C shell lists out the last few commands (the precise number being controlled by the **history** environment variable). Each command is listed with an event number. For example:

```
9 vi chapter.6
10 wc -l chapter.6
11 l ch*
12 diff chapter.6 chapter.6.old | more
13 vi chapter.7
```

To recall a given event, type an exclamation mark followed by the event number; for example, to edit *chapter.6* again, the command would be **!9**.

To alter a command, you can follow the event number with a colon (:) and a modifier which is applied to the word.

For example, to edit *chapter.8* using the above history, you could issue the instruction **!9:s/8/6/**, which substitutes "8" for "6".

A full list of modifiers is provided in the **csh**(C) manual page.

Using aliases

In the context of the shells, an alias is a specially defined synonym for a command or commands. You can define aliases for complex operations, or rename commands (for example, to make life easier if you are moving to the SCO OpenServer system from DOS).

Aliases are not available in the Bourne shell. This discussion covers the mechanisms provided by the Korn shell: the C shell also provides aliases, but the syntax differs.

Suppose you frequently need to issue the following command:

grep red | grep -v brown

This command searches a file or an input pipe for lines containing the word "red", then excludes lines that also contain the word "brown". From inside the Korn shell, you can set up an alias called **prism** that is an abbreviation for this pipe with the following command:

alias -x prism='grep red | grep -v brown'

When you next type the command **prism**, the shell will check its internal table of aliases, and replace the word **prism** with the value of the alias. So if you type the following:

cat foo | prism

The command that actually gets run is as follows:

cat foo | grep red | grep -v brown

The **-x** option to **alias** makes the alias remain in force for all scripts that run under the current shell session. Otherwise, the alias will not be exported to any shell scripts you run.

The syntax of **alias** differs under the C shell. The equivalent command to create the alias under **csh** is:

alias prism='grep red | grep -v brown'

If you use aliases a lot, you might want to save them in a file called *.aliases*, executed from your *.profile* or *.login* scripts. For example, you could add the following line to your *.login* file:

```
.  ~/.aliases
```

If you need aliases to be exported to scripts running under the current shell session, they should be defined using the **-x** option to the **alias** command (under the Korn shell).

How aliases are executed

When executing a command line, the Korn shell checks each word in turn against its table of known aliases. If the word is an alias name, and is not quoted in any way, and the shell is not already processing an alias of that name, then the alias name is replaced by the value of the alias. The process stops after the shell detects and substitutes one alias, unless the alias is followed immediately by a space. A similar process occurs under the C shell, except that aliases can explicitly refer to the history list (to recall a previously issued command).

It is possible to embed aliases so that an alias definition includes a command that is itself an alias. For example:

```
$ alias dir='ls -al'
$ alias count='dir | wc -l'
$ count
34
$
```

Here, the aliased command **count** uses the alias **dir** to list (in full) the files in the current working directory; the output is piped into **wc** to give a line count, thus indicating the number of files in the directory.

The shells provide a quoting mechanism that can be used to prevent commands being evaluated under some circumstances.

The shells keep track of alias expansion; an alias which contains its own name will only be expanded once, and aliases are not expanded if they are quoted. This prevents the shell from getting trapped in an infinite loop if it expands an alias that refers to itself.

Note that there are some drawbacks to using aliases. It is easy to accidentally redefine standard commands so that they act in nonstandard or unexpected ways. When you define an alias, it is important to make sure that no program with the same name as the alias already exists; otherwise the alias will be substituted for it. For an example of a dangerous alias (that you should not use) **alias -x kill='rm'** could have unexpected results if you were to subsequently try to **kill** a runaway process.

Aliases can include references to variables, but note that you should enclose the alias definition in single quotes to prevent the variable from being expanded as the alias is defined. For example:

> **alias random='echo $RANDOM'**

This command evaluates the variable **RANDOM**.

> **alias random="echo $RANDOM"**

In this example, the alias would have been defined with whatever the *literal* value of **RANDOM** was at the time. This is because the shell expands variable references found in double quotes, but not in single quotes. Whenever the variable **RANDOM** is referenced, it returns a different (random) number. So we would see something like the following:

```
$ alias random='echo $RANDOM'
$ random
56302
$ random
17094
$ alias random="echo $RANDOM"
$ random
12916
$ random
12916
$
```

You may encounter problems when referring to positional parameters in aliases. Positional parameters are the arguments you specify after a command name; for example, when specifying a filename as a parameter to a command. They are positional in the sense that you identify them as variables by their position along the command line. **$0** refers to the name of the program; **$1** is the first positional parameter, and refers to the first argument, **$2** refers to the second argument, and so on. See "Creating a shell script" (page 246) for more information about positional parameters.

If you use a positional parameter in an alias, the alias will expand the positional parameter of the currently running shell. Aliases are not separate programs and do not have parameters; they are simply replaced with the appropriate string on the command line. For example (in the Korn shell):

```
$ set -- bill ted mary
$ alias args='echo $3 $2 $1'
$ args foo bar quux
mary ted bill foo bar quux
$
```

The command **set --** sets the positional parameters of the shell. What is happening is that the alias is expanded to the following:

```
echo $3 $2 $1 foo bar quux
```

The first three names are positional parameters that refer to the first three arguments to the current shell; the additional names are simply tagged on to the end of the **echo** statement.

If you need to modify the Korn shell positional parameters, use the **set --** command. The arguments to **set --** are used to replace the shell positional parameters **$1, $2 ... $n.**

If you want to define a command within the shell that accepts a parameter, you must define a function. A function is a block of commands that are referred to by a name, and that take positional parameters. For example:

```
del( )
{
   rm -i "$@"
}
```

When you type **del** *file1 file2* the command line calls the function **del( )**. The shell executes the instructions defined in **del( )**, then resumes execution where it left off: in this case, back at the shell prompt. (You can also use functions in shell scripts.) Parameters to **del** are passed in the shell positional parameters **$1, $2**, and so on. **$@** is a special parameter consisting of all the positional parameters presented as a single string; so referring to this parameter allows us to interactively remove a variable number of files with one **del** command. (See "Passing arguments to a shell script" (page 250) for more information on special parameters.)

How the shell works

Your login shell reads its standard input from your terminal, and sends its standard output and standard error back to your terminal unless you tell it to send them elsewhere. (See "More about redirecting input and output" (page 256) for more information on these streams.) The shell is *line oriented*; it does not process your commands until you press ⟨Enter⟩ to indicate the end of a line. You can correct your typing as you go. Different shells provide different facilities for editing your commands, but they generally recognize ⟨Bksp⟩ or ⟨Del⟩ as the keystroke to delete the previous character.

When you press ⟨Enter⟩, the shell interprets the line you have entered before it executes the commands on that line. The steps it runs through are as follows:

1. The shell splits the line into *tokens*. A token is a command, variable, or other symbol recognized by the shell. It continues to build up a sequence of tokens until it comes to a *reserved word* (a shell internal command that governs the flow of control of a shell script), function name, or *operator* (a symbol denoting a pipe, a logical condition, a command separator, or some other operation that cannot be carried out until the preceding command is evaluated).

2. The shell organizes the tokens into three categories:

   – I/O redirection; commands that determine where the input or output of a program are directed. For example, in the following command line, the text ">listfile" is interpreted as an output redirection, which is later applied to the preceding command:

   ls -al >listfile

   – Variable assignment; the shell can recognize commands that assign a value to a variable.

   – Miscellaneous commands; other tokens are checked to see if they are aliases. The first word is checked. If it is an alias, it is replaced by the original meaning of the alias; if it is not an alias, or if it is followed by a whitespace character before the next word, the process of alias checking is repeated until no more words remain (or until an alias has been detected that is not followed by a space).

3. The commands may then be executed, either as internal shell commands (that cause the shell itself to take some action) or, if they are not internal commands, as external programs (if the shell can locate an executable file of that name).

How the shell executes commands

When the shell has processed a command line and is left with a name that is not a built-in command, the name of a function, or a second or subsequent command in a pipeline, it checks the directories listed in your **PATH** environment variable for a file that matches that name and on which the executable permission is set for users in your category.

If such a file exists and is an executable binary file (that is, a program that has been compiled into machine code), the shell *forks*; that is, it creates a copy of itself (a "child" process) in the computer's memory (see **fork**(S) for a list of the characteristics inherited by a child process from its parent).

The child process then *execs* the binary file; that is, it loads a copy of the binary file's instructions in place of its own, and begins to execute it. When a shell process execs another process, the new process completely replaces the shell process in the computer's memory. The parent shell remains in memory, and waits until the child process terminates before it resumes operation.

If the file that the shell finds is not a binary file, a different course of events occurs. The shell forks a child shell that automatically opens the file and begins to interpret it, one line at a time, as if each line is being typed on the shell's standard input. This is why such a text file is called a shell *script*; it is literally a script of actions to be carried out by the subshell.

Note that the output of a script that runs in a subshell is not automatically available to the parent shell; while the subshell "inherits" (that is, receives a copy of) the environment of its parent, the parent does not experience any changes that the subshell makes to its environment. So if you use a script to set a variable, the variable will not be present in the parent shell's environment.

You can work around this by using the dot (.) command. If you have a script called **myprog**, then you can execute it as follows:

. **myprog**

The script will be opened and interpreted directly by the current shell, without forking a sub-shell. Therefore, a script executed by the dot command can change your environment; a script executed by typing its name cannot.

For example, suppose you have a script called *setvars*, which contains the following:

```
PATH=/bin:/usr/bin:/usr/local/bin:$HOME/bin:
```

You can use it to change your path only if you execute it with the dot command. For example:

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
$ setvars
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
$ . setvars
$ echo $PATH
:/bin:/usr/bin:/usr/local/bin:$HOME/bin:
$
```

Chapter 11

Automating frequent tasks

This chapter explains how to write shell scripts to automate repetitive tasks. It describes how to:

- create a shell script (page 246)
- send messages to a terminal (page 253)
- receive input from a file or terminal (page 259)
- solve problems (page 263)

- tune script performance (page 291)

- control program performance (page 291)

Shell scripts are useful when there are two or more commands that you frequently run at the same time, or when there is some complex task that you want to automate. For example, you can use shell scripts to keep watch on your mailfolders and prepare various reports on their contents; or you can use shell scripts to automate backup procedures, periodically copying files to tape. However, the shells provide such a powerful tool that complex programs have been written using the scripting language: the uses are limited only by your imagination.

The syntax described in this chapter is common to the Bourne shell and the Korn shell; for C shell syntax, refer to **csh**(C) in the *Operating System User's Reference*.

Creating a shell script

A shell script is a text file containing a sequence of shell commands. The commands are normally entered on separate lines, for readability, but can be separated by semicolons (;).

To create a shell script, create a new file with a text editor (for example **vi**) and type SCO OpenServer commands into it. Save the file, and use **chmod** to set the *executable* permission bit on the file so the shell can run it. For example:

    vi is.logged.in

Enter the following text:

    who | grep fred

Save the file, and issue the following command:

    chmod +x is.logged.in

This assigns the owner of *is.logged.in* permissions to read, write, and execute the file.

If the current working directory is included in your search path (**$PATH**), you can execute the file as follows:

```
$ . is.logged.in
fred console Aug 13 11:28
```

If the file is not held in a **PATH** directory, an alternate notation is required:

```
$ ./is.logged.in
fred console Aug 13 11:28
```

The notation "./" has the same effect as typing the directory's absolute pathname.

The program runs the command **who**, to list currently logged in users on the system, then uses **grep** to search it for the line containing fred, indicating that fred is logged in.

Suppose you want to use the script to see if people other than **fred** are logged in. You can modify the script as follows:

```
who | grep $1
```

The positional parameter **$1** (used in place of **fred**) refers to the first word on the command line after the name of the script. Where **$1** is used in the script, it is substituted for the first argument entered on the command line. You use it like this:

```
$ ./is.logged.in fred
fred console Aug 13 11:28
$ ./is.logged.in mary
$
```

fred is logged in; **mary** is not logged in. This script gives no output if it cannot find the name you supply it with in the output from **who**.

Running a script under any shell

You can make a shell script execute under any given shell, even if that shell is not currently running or is not the shell the script was written for, by placing on the first line of the script the special command **#!***shellpath*, where *shellpath* is the absolute pathname of the shell under which the script is to execute.

For example, if your login shell is the C shell, but you want to write scripts for the Korn shell, the first line of your script should be as follows:

```
#!/bin/ksh
```

This is a general mechanism: that is, *shellpath* does not have to be a shell, but can be any binary program. For example, **awk**(C) is a small programming language used for textual analysis tasks (see Chapter 13, "Using awk" (page 323) for an introduction to using **awk**). **awk** scripts could start with the following:

```
#!/usr/bin/awk -f
```

If the **-f** flag is omitted, **awk** will exit with an error. See **exec**(M) for details of this mechanism.)

Writing a short shell script: an example

The SCO OpenServer system identifies and manages files and directories using the concept of the *inode* (index node). Each inode has a unique numerical identifier and stores such information as the file type, its size, where it is physically stored on disk, the date and time of the most recent access and modification, and so on. The filename is simply an aid for the user, and each file can have more than one filename. Each pathname to a file is known as a link. (For information on linking files, see "Creating links to files and directories" (page 94).)

When a file has several links, you must remove all of them before you can delete the file itself. Therefore, it is desirable to be able to trace all the links to a given file. You can do this using the inode to search for all the filenames that have that inode number.

To list the inode of a file, use the following command:

ls -i

This lists each inode in the current directory, followed by the filename associated with it:

```
$ ls -i
 1125 0.order.number
 2852 0.parts.index
 5315 0.order.index
  770 00.partno.err
 4225 00.partno.out
$
```

For example, the inode number of *0.parts.index* is 2852.

To find the inode of a given file, you could type something like the following:

ls -i 0.parts.index | awk '{print $1}'

The first part of the pipeline lists the inode of the file called *0.parts.index*. The output from this command is fed into **awk**, which prints the first field, that is, the inode number:

```
2852
```

However, printing the inode number of a file and using the inode number to do something useful are not the same. We need some way to capture the output of a command.

This can be done using a variable, using the backquote notation (recognized in the Bourne and Korn shells):

variable=`command`

command is executed, then its output is stored in *variable*. For example:

```
$ myinum=`ls -i 0.parts.index | awk '{print $1}'`
$ echo $myinum
2852
$
```

The C shell recognizes the corresponding notation:

set *variable `command`*

Having obtained the inode number and stored it in an environment variable, we can then use it in a **find** command. For example:

find / -inum $myinum -print

The **find** option **-inum** tells **find** to look for files matching the inode number stored in the variable *myinum*. The option **-print** tells **find** to print the names of any files it matches. (This command also outputs a list of all the directories it cannot access as it reaches them.)

Now we can write a shell script that, given a filename, searches for all links that point to the same file:

```
myinum=`ls -i $1 | awk '{ print $1 }'`
find / -inum $myinum -print 2> /dev/null
```

In summary, the first line assigns the inode of the specified file (here represented by the positional parameter *$1*) to the variable *myinum*. Note that the second "*$1*" notation in this line is internal to **awk**, and refers to the first field of the output piped from **ls -i**, and not back to the specified filename.

The second line of the script invokes **find**, telling it to start at the root directory (/) and search through all the mounted filesystems for files matching the inode number found in the variable **myinum**, and print their names.

Note that an inode number is only unique within a given filesystem. It is possible for two files with the same inode number to exist on different filesystems, and not be linked together. It is therefore worth checking the output to make sure that all the files output by this script reside on the same filesystem.

find prints a message to the standard error if it cannot look inside a directory. We do not want to see these error messages, so the *standard error* output from **find** (output stream 2) is redirected to the device */dev/null*; an output stream sent to this device is silently ignored. Consequently, the error messages are discarded and a clear, uncluttered output is produced. (Non-spurious errors are also indiscriminately discarded. However, in this example all errors are probably spurious, so discarding all messages is acceptable.)

Passing arguments to a shell script

Any shell script you run has access to (inherits) the environment variables accessible to its parent shell. In addition, any arguments you type after the script name on the shell command line are passed to the script as a series of variables.

The following parameters are recognized:

$*   Returns a single string ("$1, $2 ... $n") comprising all of the positional parameters separated by the internal field separator character (defined by the **IFS** environment variable).

$@   Returns a sequence of strings ("$1", "$2", ... "$n") wherein each positional parameter remains separate from the others.

$1, $2 ... $n   Refers to a numbered argument to the script, where *n* is the position of the argument on the command line. In the Korn shell you can refer directly to arguments where *n* is greater than 9 using braces. For example, to refer to the 57th positional parameter, use the notation ${57}. In the other shells, to refer to parameters with numbers greater than 9, use the **shift** command; this shifts the parameter list to the left. **$1** is lost, while **$2** becomes **$1, $3** becomes **$2**, and so on. The inaccessible tenth parameter becomes **$9** and can then be referred to.

$0   Refers to the name of the script itself.

$#   Refers to the number of arguments specified on a command line.

For example, create the following shell script called *mytest*:

```
echo There are $# arguments to $0: $*
echo first argument: $1
echo second argument: $2
echo third argument: $3
echo here they are again: $@
```

When the file is executed, you will see something like the following:

```
$ mytest foo bar quux
There are 3 arguments to mytest: foo bar quux
first argument: foo
second argument: bar
third argument: quux
here they are again: foo bar quux
```

$# is expanded to the number of arguments to the script, while $* and $@ contain the entire argument list. Individual parameters are accessed via $0, which contains the name of the script, and variables $1 to $3 which contain the arguments to the script (from left to right along the command line).

Although the output from $@ and $* appears to be the same, it may be handled differently, as $@ lists the positional parameters separately rather than concatenating them into a single string. Add the following to the end of *mytest*:

```
function how_many {
        print "$# arguments were supplied."
}
how_many "$*"
how_many "$@"
```

The following appears when you run *mytest*:

```
$ mytest foo bar quux
There are 3 arguments to mytest: foo bar quux
first argument: foo
second argument: bar
third argument: quux
here they are again: foo bar quux
1 arguments were supplied.
3 arguments were supplied.
```

Performing arithmetic and comparing variables

It is sometimes useful to perform arithmetic, compare variables or check for the existence of files using the shell. There are four ways to do this:

- Use the **test**(C) program to test variables for equivalence or existence.

- Use the **expr**(C) expression evaluator to calculate on variables (as integer numbers) or compare variables (as strings of text).

- Use the **bc**(C) binary calculator (or another calculator, such as **dc**(C) or **awk**) to carry out more complex mathematical operations (on decimals, fractions, and unusual bases).

- Under the Korn shell only, use the ((..)) notation to evaluate simple mathematical operations. This notation is equivalent to **let "..."**. Note that the ((..)) test is built into the shell, and therefore executes faster.

test allows you to check if a named file exists and possesses some property, or to test whether two strings are similar or different. **test** is explained in detail in "Different kinds of test" (page 277).

expr evaluates an expression and prints the result, which can then be captured with backquotes. For example:

```
$ var=65
$ result=`expr $var \* 5`
$ echo $result
325
$
```

Note the backslash in front of the " * " symbol. * is short for multiplication in **expr** (and many other programs), but the shell treats it as a filename wildcard character and replaces it with a list of matching files unless it is escaped (see Chapter 12, "Regular expressions" (page 315)).

expr can also be used to manipulate variables containing text (strings). A portion of a text string can be extracted; for example:

```
$ expr substr bobsleigh 4 6
sleigh
$
```

The **substr** expression returns a substring of its first parameter ("bobsleigh") starting at the character position indicated by its second parameter (the fourth character: the character is "s"), of a length indicated by its third parameter (6 characters).

There are many additional options to **expr**. In general, you can use **expr** to search a string for a substring, extract substrings, compare strings, and provide information about a string. It can also perform basic arithmetic on integer numbers, but not on real numbers. For calculations that require decimals or fractions, you should use a calculator, like **bc**. (See "Putting everything together" (page 298) for an example of using **bc** within a shell script.)

Performing arithmetic on variables in the Korn shell

The Korn shell can be told to perform arithmetic using variables. Because this facility is built into the shell calculations can be executed faster than by using **expr**, which is a separate program that must be forked and exec'ed (see **fork**(S) and **exec**(S)).

Although variables are normally treated as strings of characters, the command **typeset -i** can be used to specify that a variable must be treated as an integer, for example **typeset -i** MYVAR specifies that the variable MYVAR is an integer rather than a string. Following the **typeset** command, attempts to assign a non integer value to the variable will fail:

```
$ typeset -i MYVAR
$ MYVAR=56
$ echo $MYVAR
56
$ MYVAR=fred
ksh: fred: bad number
$
```

To carry out arithmetic operations on variables or within a shell script, use the **let** command. **let** evaluates its arguments as simple arithmetic expressions. For example:

```
$ let ans=$MYVAR+45
echo $ans
101
$
```

The expression above could also be written as follows:

```
$ echo $(($MYVAR+45))
101
$
```

Anything enclosed within **$((** and **))** is interpreted by the Korn shell as being an arithmetic expression. It is possible to include variables within such arithmetic expressions; it is not necessary to prefix them with the usual dollar sign although no error condition is caused if the dollar sign is used.

If you need to carry out calculations on floating point numbers, it is necessary to use the binary calculator, **bc**.

Sending a message to a terminal

There are several methods of producing output in a shell script. The first, and simplest, is the **echo** command used in the last example (see "Passing arguments to a shell script" (page 250)).

Note that the **echo** command exists in four separate forms. Originally, **echo** was a separate program, **/bin/echo**: but a version of it is now built into all three shells. There are subtle differences between them, and although the core functionality is the same (the command **echo hello** always prints the word "hello") you should check any special options you use against the relevant shell manual pages. Next, the Korn shell provides the **print** command. **print** is more versatile than **echo**, but cannot be used under the Bourne shell.

Finally, a more sophisticated output mechanism is the **printf** command. This is similar to the **printf** command built into **awk** and the callable function used by the C programming language. See **printf**(C) for details.

As far as the system is concerned, terminals are just a special type of file. You send data to a terminal or read data from it just like any other file.

The echo command

The **echo** command prints its argument list, separating each argument with a space, and following the last argument with a newline. For example:

```
$ echo Hi there!
Hi there!
$
```

Variables and file specifications are expanded by the shell before being passed to **echo**. Consider the following command:

echo The available files are *

This prints the specified text string before producing a listing of all the files in the current working directory, across the screen.

echo recognizes a number of *escape sequences* which it expands internally. An escape command is a backslash-escaped character that signifies some other character. The ones recognized by **echo** are common throughout the shell syntax, as follows:

\a   Alert (rings the terminal bell)

\b   Backspace

\c   No newline at end of **echo** output

\f   Form feed

\n   Newline

\r   Carriage return (no newline)

\t   Tab

\v   Vertical tab

char   Quotes a character with special meaning to the shell. For example, "\\" generates a single backslash: as an escape character, the backslash must be escaped or quoted to stop the shell processing it as the prefix to a command.

nnn   *nnn* is an octal number, exactly three digits long, which represents an ASCII character value to insert.

Note that one of the quoting mechanisms must be employed when using escape sequences with the **echo** command, as follows:

```
$ echo The available files are \n *
The available files are
aaaa bbbb cccc dddd eeee
```

Here, the escape sequence only is quoted. Otherwise, the whole string can be quoted:

```
$ foo="a\ty"
$ echo $foo
a       y
$
```

For example, see the following **echo** command:

```
$ echo "Mary had a little lamb \n \t Its fleece was white as snow"
Mary had a little lamb
            Its fleece was white as snow
```

The \n escape causes **echo** to emit a newline, and the \t escape causes **echo** to emit a tab.

You can redirect the output from **echo**. For example, the **who** and **w** commands list the users on your system and the terminals they are logged in on. To send a message to a terminal being used by someone else, you can use a command like the following, if */dev/tty015* is the name of the terminal you want to print a message on:

```
$ echo Hi there! > /dev/tty015
```

(Note that this is not the best way to send messages between terminals; **write**(C) and **talk**(TC) are commands intended for this purpose, and allow two-way conversation.)

The print command (Korn shell only)

In the Korn shell, **print** is preferred to **echo**. **print** is built in to the shell and behaves just like **echo** and recognizes the same escape commands. It also accepts the following options:

-    Anything following the - is processed as an argument, even if it begins with a -.

-R   The escape conventions (commands beginning with \) are ignored. Anything following the **-R** (except a **-n**) is treated as an argument, even if it begins with a " - ".

-n   **print** does not append a newline to its output.

-p   If you have started a co-process running with the | & command (see "More about redirecting input and output" (page 256)), the **-p** flag makes **print** send its output to the co-process via a pipe.

-r   **print** ignores the - escape commands and prints their literal value (that is, a backslash followed by the escape command letter).

-s   **print** sends its output to the history file. This enables you to add commands to your history file from a shell script without executing them; you can subsequently recall or edit them rapidly, without needing to retype them.

-u*n*   **print** sends its output to file descriptor *n*.

The **-u** option is equivalent to redirecting the standard output, but doesn't open or close the destination file. This is particularly useful if you have opened some files in **ksh** and want to write data to them (for later reading with the **read** command); see "More about redirecting input and output" (this page).)

More about redirecting input and output

A program running under the shell can have several files open to it simultaneously for reading and writing. They are identified by their *file descriptors*, numbers used by the system to associate a file with an input or output stream. The system treats each open file as a stream of characters that flow sequentially, from start to finish. The streams associated with any program are the *standard input*, represented by file descriptor 0, the *standard output* (file descriptor 1) and the *standard error* (file descriptor 2).

Basic shell syntax

The basic shell syntax for redirecting input and output is as follows:

<*file*   Use *file* as a source of standard input.

<*nfile*   Read *file* as a source of input to file descriptor *n*.

>*file*   Write standard output to *file*.

n>*file*   Write the output from file descriptor *n* to *file*.

In the following example, the file called *thing* does not exist:

```
$ cat thing
cat: cannot open thing: No such file or directory
$ cat thing 2> /dev/null
$
```

This effect is particularly useful when appended to a command that generates copious but unwanted error messages; it sends the output from file descriptor 2 (the standard error) to */dev/null,* the "bit bucket" or "black hole" device. (*/dev/null* is also known as the null device; if you send data to it, it absorbs it silently, and if you read from it all you get is a null character.)

Other useful fragments are:

>&2   appended to an echo, sends the output to the standard error

2>&1   merges the standard error with the standard output

Note that when the "`>`" symbol is employed, the file it is directing output to is either created or, if it already exists, is erased and replaced. This is known as "clobbering" a file. (The system knows better than to destroy terminal or tape special device files this way: the tape or screen controlled by the device is overwritten, but the device file itself in */dev* is not affected.)

To append output to the end of an existing file, use the "`>>`" notation instead.

If you want to permanently prevent the Korn shell from destroying an existing file when you use the "`>`" redirection operator, adjust the shell parameter **noclobber** by issuing the command **set -o noclobber**. If the shell finds that a file it is writing to already exists, it will issue an error message and refuse to overwrite it, as follows:

```
$ cat aaa > bbb
ksh: bbb: file already exists
```

Once **noclobber** is set, you have to redirect using the override command, **>!** (instead of >) if you want to overwrite it.

The << operator has a special meaning: it is used to tell the shell to read its standard input from the current script. For example, if you have a shell script containing the line:

```
<<terminating_string
  :
  :
```

Everything from that line down, until it encounters a line with just "terminating_string" on it will be taken as a *here document*, a file which is treated as the standard input. So, to send a multiline message to the screen, instead of using **print** or **echo** you could embed a help message in your script:

```
_help( )
{
  cat <<%%

  Readability Analysis Program

  A shell/awk demo to determine the readability grade of texts

  Either run rap with no options for full menu-driven
  activity, or use the following flags:

  -[h|H]   prints this help
  -l       cause output to be logged to a file
  -f file  enter the name of the file to check
  -b       run in batch mode (no menus)

%%
exit 1
}
```

This defines a function called **_help** within a shell script. When the script subsequently encounters the command **_help** it will **cat** the text between two sets of "%%" symbols to the standard output, then exit.

Scripts running under the shell may have many file descriptors in use simultaneously. Some programs may not be able to deal with reading and writing lots of redirected file descriptors: other programs expect to read a filename on their command line, rather than look for redirected input.

To get round this, you can use the special files */dev/stdin*, */dev/stdout*, and */dev/stderr*; see "Forcing a program to read standard input and output" (page 119) for an example of this.

The following example shows an instance of extracting streams of information from one file and placing them in two different output files using only one pipeline, as follows:

2>second_field; cat myfile | awk '{ print $2 > "/dev/stderr"; print $1 }' | sort

The first command on this line attaches the standard error to a file (in this instance *second_field*). The input file *myfile* is then piped into an **awk** program. The **awk** program prints the second field of every line to */dev/stderr*, the standard error, and prints the first field of every line to the standard output. Because the standard error has been redirected, the second field of each line ends up in *second_field*, while the first fields are sorted and presented on the standard output.

Getting input from a file or a terminal

In addition to printing information on the screen and redirecting the output from commands, you will almost certainly want to let your scripts prompt you for information, and make use of that information. The Bourne and Korn shells both provide the **read** command, which is the inverse of **print** or **echo**; it reads a line from a file (using the standard input as a default) and stores the successive words in the line in a series of shell variables which you specify on the command line. If you don't specify enough variables to hold all the words on the line read by **print**, all the remaining words will be stored in the last variable you name.

For example, suppose we use the following script to get a line of input from the terminal:

```
print Hi there! Please type a line of text.\n
read foo
print $foo
```

When you run the script, it prompts for a line of text, and **read**s it all into the variable **foo**. The next line then prints the contents of **foo**. (Remember, to the shell, **$foo** means "the contents associated with the variable named **foo**", but **foo** on its own is simply a name; so the command **print foo** will output the word "foo", rather than the contents of the variable **foo**. This is a common pitfall when you start programming the shell.) For example, if the script above is called *getline*:

```
$ ./getline
Hi there! Please type a line of text.
This is a test.
This is a test.
$
```

The Korn shell provides a shorthand notation for this, as follows:

```
read 'foo?Hi there! Please type a line of text. '
```

This is equivalent to the following:

```
print Hi there! Please type a line of text.
read foo
```

Text up to the question mark is interpreted as the name of a variable in which the input is stored: text after the question mark is used as a prompt.

To read two words into different variables, you might use a script like the following:

```
print Hi there! Type two words then press enter.\n
read foo bar
print The first word I read was $foo
print and the second was $bar
```

If you type *three* words when you run this script, instead of two, the last two words will appear in the second variable. For example, if the script is called **getwords**:

```
$ ./getwords
Hi there! Type two words then press enter.
hello yourself, program!
The first word I read was hello
and the second was yourself, program!
$
```

When you use the Korn shell (but not the Bourne shell) **read** takes a number of options. These are as follows:

-p   Read input from a co-process. The shell disconnects from its pipe to the co-process when an error or end-of-file condition is read.

-s   Save the input line as a command in the history file (without executing the command).

-u*n*   Read a line from file descriptor *n*. The default is file descriptor 0, the standard input.

For the other options and the arguments to **read**, refer to **ksh**(C).

Reading a single character from a file or a terminal

read reads a line of text at a time, but it is often useful to have a script wait for a keystroke, then act on that keystroke immediately. For example, when using a menu driven program, you may not want the program to wait for you to press ⟨Enter⟩ after you select an item. There is no command to obtain a single character from a terminal, but we can simulate one.

Here is a simple function to obtain a keystroke:

```
getc ( )
{
  stty raw
  tmp=`dd bs=1 count=1 2>/dev/null`
  eval $1='$tmp'
  stty cooked
}
```

To use it, insert it at the top of your shell script, then invoke it lower down the shell script:

```
echo "Enter a character: \c"
getc char
echo
echo "You entered $char"
```

getc puts the terminal into *raw* mode. Instead of passing your input through to the system a line at a time, the terminal now passes each keystroke you type straight through, unmodified.

The **dd** command reads a single character from the standard input and writes it to the standard output, that is captured in the variable **tmp**. The next line is used to assign the literal contents of **tmp** to the variable named by **$1**. The **eval** command in front of this line is necessary to force the shell to scan the line twice; once to expand **$1** into the name of a variable, and again to carry out the actual command. The quotes around **$tmp** are stripped off by **eval**; if you omit them, then if your character is a whitespace character, it will be lost.

Afterwards, **getc** puts the terminal back into normal operating mode with the command **stty cooked** (or **stty -raw**, or **stty sane**).

We can write **getc** more succinctly like this:

```
getc ()
{
  stty raw
  eval $1=´`dd bs=1 count=1 2>/dev/null`´
  stty cooked
}
```

Because **getc** returns a single character in whatever variable you specify, you can use it flexibly. For example, the following function can be used to make a program pause until you are ready for it to continue:

```
press_any_key()
{
  echo "Strike any key to continue ...\c"
  getc anychar
}
```

Combine the two functions in a script called *char_handler*, as follows:

```
getc ()
{
  stty raw
  eval $1='`dd bs=1 count=1 2>/dev/null`'
  stty cooked
}
press_any_key()
{
  echo "Strike any key to continue ...\c"
  getc anychar
}
echo "Enter a character: \c"
getc char
echo
echo "You entered $char"
press_any_key char
echo \r
```

Execute *char_handler* as follows:

```
$ ./char_handler
Enter a character: x
You entered x
Strike any key to continue ...y
$
```

Attaching a file to a file descriptor

Most of the time, you will only need to work with three file streams; the standard input, standard output, and standard error. However, if you need to read input from a file into a shell script, or to send output to one or more other files, you may want to open some more files and attach them to file descriptor numbers.

To open files for reading, use the **exec** command. **exec** causes the commands following it on the line to be executed immediately without invoking a subshell. The command to be **exec**ed overlays the shell process, and when it terminates control returns to the *parent* of the process that carried out the **exec**.

You can use **exec** to attach new files to the input and output file descriptors of the current shell process. For example, to open a file called *newscript* as standard input to the current shell, use the following command:

exec <newscript

newscript should be executable and contain the following line:

```
echo "Hello world!"
```

In this case, **exec** forces *newscript* to be opened as standard input, then causes its contents to be executed.

To open *file1*, *file2* and *file3* for input as file descriptors 1, 4 and 5 respectively, use the following:

> **exec 1< file1 4< file2 5< file3**

Note that there is an anomaly in the Korn shell when opening file descriptors using **exec**. Although the Bourne and Korn shells allow you to open any recognized file descriptor for input or output, the Korn shell closes them immediately after executing the command line (with the exception of file descriptors 0, 1 and 2: standard input, standard output, and standard error). The C shell does not allow you to redirect or attach file descriptors: this is one of its major shortcomings.

What to do if something goes wrong

If your shell script stubbornly refuses to work, there are two possibilities:

- You are trying to execute the script in an inappropriate environment.

- The script contains a bug.

An inappropriate environment means that the script is unable to run because the environment you are trying to run it in is not set up for it. For example, you cannot execute a Bourne shell script in the C shell with any expectation of success (unless you force the system to run the script under a Bourne shell by making the first line of the script **#!/bin/sh**). Alternatively, you may have forgotten to set the execute permission on the script, so that the shell fails to recognize it as a command. Or you may have told your script to read and act on an environment variable which is not present.

Solving problems with the environment

A particularly common error is to fail to include " . " (the current working directory) in your **PATH** variable. (Note that **PATH** recognizes a colon with no trailing characters, or a colon followed immediately by another colon and a pathname, as synonyms for " . ".) When the shell reads a command name it only searches for an executable file of that name in the directories listed in **PATH**. If " . " is not included in **PATH**, the shell will not look for the file in your current directory. Including "." in **PATH** removes the necessity of ever having to use the "./" notation to execute your scripts (see "Creating a shell script" (page 246)).

Another common error is to give your file the same name as an existing command. If the current directory (.) precedes the directory in which the synonymous command exists in your **PATH**, your script will be used instead of the command whenever you call it; on the other hand, if the directory in which the command exists is before " . " in your **PATH**, the command will be executed instead of your script.

Consider the following search path:

> **/bin:/usr/bin:/u/charles/bin::/usr/sco/bin:/u/bin:**

For example, if you create a script called *test* in the current directory, and you attempt to execute it by typing the command **test**, the shell will search along your path and execute */bin/test* instead of *./test* (pointed to by the fourth, null, field in the path).

Try to avoid giving your scripts a name already used by a SCO OpenServer utility. A quick way to test a proposed name is to invoke **man** on it; if **man** provides a manual reference, it is a bad idea to use the name. It is also worth checking the relevant manual page for the shell you are using, in case your script shares a name with a built in shell command.

Another common problem is to invoke a script under the wrong shell. To ensure that the script is always run by the correct shell, use the hash-bang notation (#!) on the first line of the script to specify which shell to use (See "Running a script under any shell" (page 247)).

Solving problems with your script

Even if your environment is set up correctly, any long script that you write will almost certainly fail to work correctly under some circumstances. This may be due to a failure to consider all the conditions under which the script may be run, or due to an oversight or syntax error in the script. The best way to get used to creating small to medium sized shell scripts is to do the following:

- Work out what you want the script to do.

- Decompose the successive stages in the process into separate steps.

- Test and debug each individual step interactively, at the shell prompt.

This method, known as bottom-up programming, is especially suited to small scripts (those which contain less than about fifty lines of commands). For longer programs, you may need to learn more about programming techniques. (See "Learning the shells" (page 428) and "Learning the C programming language" (page 428) for references to more advanced texts.)

What to do if your shell script fails

In the meantime, if your shell scripts fail, a useful technique for finding out what is going wrong is to use the **-x** flag. You can set it when you start **ksh** by running the shell with the command **ksh -x**; or you can set it from within the Korn shell by issuing the following command:

> **set -o xtrace**

The Bourne shell's equivalent is as follows:

> **set -x**

The **xtrace** option causes the Korn shell to list each command after it has been expanded, but before it has been executed. This enables you to catch any errors due to alias substitution, wildcard expansion, or quote stripping.

(The **set -o** command can be used to reset the Korn shell's startup options from within a running shell; type **set -o** for a listing of the current option states, then use **set -o** *option* to switch *option* on, or **set +o** *option* to switch *option* off.) The **set -** command will also turn off the **xtrace** facility.

Another useful technique is to use **print** as frequently as possible, to let you know what your script thinks it is meant to be doing. Print the contents of variables before and after you change them, along with a message to explain what kind of operation you are carrying out. Better still, make **print** send this output to a log file. The file provides you with a permanent record of what happened during a test run of the script.

An important rule to bear in mind if your script fails is not to change more than one thing at a time between test runs. Errors are eliminated by making a single change to a script, running it, and seeing how it behaves, then trying to deduce where the error is coming from. Randomly changing your script will make it much harder to pinpoint the source of errors and is unlikely to eliminate them.

Here is an extended example that demonstrates these techniques and introduces some new concepts.

Writing a readability analysis program: an example

For the rest of this chapter, and at intervals in the following chapters, we will refer to a single recurring example: a program to analyze the readability of text files. Such a program needs to identify the files it is to work on. It must open them, use several other programs to obtain information about the files, then print the results. It also serves as a demonstration of several useful techniques: notably, how to built a simple menu driven program, how to build up complex regular expressions, and how to integrate **awk** scripts and other programming languages into shell programs.

The objective of a readability analysis program is to scan a file or files of text, and report various statistics about their internal complexity. There is more to this than just running **wc**; we want to generate a report on such things as the number of sentences in a file, the average length of each sentence, the average number of syllables per word, and the readability grade of the file. It would be useful to be able to invoke the program from the shell prompt with a variety of options: it would also be useful to provide the program with a menu driven front end. All these tasks, and more, will be explained as we encounter them in building up our example.

The first step in writing a large program is to analyze what it is intended to do: what its inputs are, and what its outputs are expected to be. We can then write a "skeleton" for it: a script that does not actually do anything to the data, but ensures that all the pieces are in place. (The actual task of analyzing a file for readability can be farmed out to a function that we will fill in later.) This is described below.

How to structure a program

In general, there are two types of program: batch programs, and interactive programs. The internal structures of batch and interactive programs differ considerably.

A batch program is a typical SCO OpenServer filter. You run it by specifying a target file (and optional flags) at the shell prompt: it runs, possibly prints messages to the standard output, and exits.

An interactive program prints a menu. You select options from the menu: the program then changes its internal state, and prints another menu, until it has assembled all the data it needs to select and execute a routine that carries out some task. It does not exit until you select a quit option from some menu.

Interactive programs are harder to write, so we will start by looking at a short batch program. An explanation of the program follows the code:

```
1 :    #!/bin/ksh
2 :    #-----------------------------------------------------
3 :    #
4 :    # rap -- Readability Analysis Program
5 :    #
6 :    # Purpose: skeleton for readability analysis of texts.
7 :    #
8 :    #------------ define program constants here ----------
9 :    #
10 :   CLS=`tput clear`
11 :   HILITE=`tput smso`
12 :   NORMAL=`tput rmso`
13 :   #
14 :   #---------- initialize some local variables -----------
15 :   #
16 :   SCRIPT=$0
17 :   help='no';      verbose=' '            ;      record=' '
18 :   log=' ' ;       next_log_state=' '     ;      batch=' '
19 :   file=' ';       fname=' '
20 :   #
21 :   #---------------- useful subroutines ----------------
22 :
23 :   do_something()
24 :   {
25 :       #   This is a stub function; it does not do anything, yet,
26 :       #   but shows where a real function should go.
27 :       #   It contains a dummy routine to get some input and exit.
28 :       echo
29 :       print "Type something (exit to quit):"
30 :       read temp
31 :       if [ $temp = "exit" ]
32 :       then
33 :         exit 0
34 :       fi
35 :   }
36 :
37 :
38 :   _help()
39 :   {
40 :     echo "
41 :
42 :           ${HILITE}Readability Analysis Program${NORMAL}
43 :
44 :           A shell/awk demo to determine the readability grade of texts
45 :
46 :           Usage: $SCRIPT -hHlb -f <file>
47 :
```

```
48 :            Either invoke with no options for full menu-driven
49 :            activity, or use the following flags:
50 :
51 :            -h or -H        prints this help
52 :            -l             log output to file
53 :            -f file        name of file to check
54 :            -b             run in batch mode (no menus)
55 :
56 :    "
57 :    }
58 :    #
59 :    #
60 :    TrapSig( )
61 :    {
62 :        echo ""
63 :        echo "Trapped signal $1...`c"
64 :    }
65 :    #
66 :    #========== START OF MAIN BODY OF PROGRAM ============
67 :    #
68 :    #------------ define program traps -------------------
69 :    #
70 :    for foo in 1 2 3 15
71 :    do
72 :      trap "TrapSig $foo" $foo
73 :    done
74 :    #
75 :    #---------- parse the command line--------------------
76 :    #
77 :    mainline=$*
78 :    echo ""
79 :    while getopts "hHvlbf:" result
80 :    do
81 :      case $result in
82 :            h|H)    help=yes                ;;
83 :            v)      verbose=yes             ;;
84 :            l)      record=yes
85 :                    next_log_state=off
86 :                    log=ON                  ;;
87 :            b)      batch=yes               ;;
88 :            f)      file=yes
89 :                    fname=$OPTARG           ;;
90 :            *)      help=yes                ;;
91 :      esac
92 :    done
93 :    shift `expr ${OPTIND} - 1`
94 :    if [ $help = 'yes' ]
95 :    then
96 :      _help
97 :      exit 1
```

```
98 :    fi
99 :    #
100 :   #---------- enter the main program --------------------
101 :   #
102 :   while :
103 :   do
104 :      do_something
105 :   done
```

(Line numbers are provided for reference only, and are not part of the program.)

At first sight this appears to be quite a complicated program, but most of it is used to set up some facilities which will be useful later. The real start of the program is line 10:

```
09 :    #------------- define program constants here ----------
10 :    CLS=`tput clear`
11 :    HILITE=`tput smso`
12 :    NORMAL=`tput rmso`
13 :    #
14 :    #---------- initialize some local variables -----------
15 :    #
16 :    SCRIPT=$0
17 :    help='no';     verbose=' '           ;      record=' '
18 :    log=' ' ;      next_log_state=' '     ;      batch=' '
19 :    file=' ';      fname=' '
20 :
```

Text following a "#" is ignored by the shell. This comes in useful when you want to leave comments in your program for other users.

Lines 10 to 20 set a number of variables. These variables are only used while the program runs: when the script ends, they will not be made available to its parent shell. One set, **CLS**, **HILITE**, and **NORMAL**, are constants; they are not changed during the execution of the program. The second set are variables that the program may use. We initialize them (to a string containing a single ⟨Space⟩ character) in case they have some other meaning within the parent shell from which the script is executed.

It is worth considering lines 10-12 in more detail. Lines of the form *variable*=`**tput** *mode*` use the command **tput**(C) to obtain the codes necessary to put the terminal into some special mode, for example reverse video mode, or to restore it to normal.

All terminals have the capability to carry out some basic actions when they receive a corresponding control code: for example, positioning the cursor, switching to reverse video, and clearing the screen. Because different terminals use different control codes, the system terminfo database maintains a table of the codes to use for a given capability on any specified terminal. These capabilities are assigned symbolic names, and the terminfo database matches the name to the escape code for each terminal.

tput takes a terminal "capability" name and returns the escape sequence to use for the current terminal. In this program, we capture the output from the **tput** command in a variable for later use. Once you have the control code for a given capability, you can **echo** the code to your terminal and it will enter whatever mode you specified.

We are using three special terminal-dependent capabilities here:

clear   Clear the entire screen.

smso   Put the terminal into reverse video mode.

rmso   Restore the terminal to normal text mode.

You can enter these modes at any time by using the command **tput** *mode*, but if you are going to use the command more than once in a shell script it is better to store the control code in a variable and echo it: this saves you from having to run **tput** every time.

Lines 23 to 56 define two functions: a stub (which does nothing useful), and a help routine. The stub simply shows where a more complex function will go, when we have written it. (At present, it prompts for an input string; if you type **exit** the script terminates.) The help routine is similar to the one we looked at in "More about redirecting input and output" (page 256). If it is called later in the script it prints a message and exits, terminating the script. Note the use of the variable **$SCRIPT** in the help function. **SCRIPT** is initialized to whatever the name of the function is, when it is executed. (It is used here in case someone renames the script, so that the usage message reflects the current name of the program.)

Note that before you can call a function, it must have been defined and the shell must have read the definition. Therefore, functions are defined at the top of a shell script and the actual program (that calls them) is right at the bottom.

Making a command repeat: the for loop

Lines 70 to 73 allow our script to survive if it receives a signal. Interactive scripts frequently do this, but batch scripts rarely do so. First, we provide a function to handle signals if any are received. It expects a parameter, **$1**, that tells it the number of the signal. All the example below does at present is to echo the number of the signal and exit, but later on we will show how it can be used to resume control of the program if something goes wrong. The signals are caught by the traps set up in lines 56 to 59:

```
for foo in 1 2 3 15
do
        trap "TrapSig $foo" $foo
done
```

This is an example of a **for** loop.

A **for** loop is a mechanism for repeating an operation for every item in a set. The general structure of a **for** loop is as follows:

```
for variable in list
do
 command
 command
    •
    •
    •
done
```

In the example, *variable* is set in turn to each value in the *list* (a collection of items from the command **in** to the end of the line). All the commands between **do** and **done** are carried out, for each successive value of *variable*. So in the example, the variable **foo** is set to 1 and the **trap** command is carried out; then **foo** is set to 2, then 3, and so on.

You can assign strings such as filenames to variables in a **for** loop. This enables you to use **for** loops to apply several commands in order to every file in a directory, or to iteratively work through a list of words (for example, invoking **mail** to send a personalized message to each of a list of recipients).

The loop in the example script from line 56 to line 59 is equivalent to writing the following:

```
foo=0
trap "Error $foo" $foo
foo=2
trap "Error $foo" $foo
foo=3
trap "Error $foo" $foo
     .
     .
     .
```

Each time the body of the loop (the part from **do** to **done**) is executed, it sets a trap for a signal (the number of which is set by the **for** statement).

You can use **for** loops with wildcards to select files. For example:

```
for target in *
do
   cp $target ../$target
   echo Copied $target
done
```

When the shell reads the first line, it expands the " * " into a list of all files in the current directory. Then, for each named file, the commands in the body of the loop are executed.

Getting options from the command line: getopts

Lines 77 to 98 of our example script illustrate a very important feature of any batch script: how to read parameters from a command line. Both the Bourne and Korn shells provide a built in command called **getopts** to read command line parameters. (Note that this should not be confused with the earlier, and obsolete, command **getopt**, which is inferior and should not be used.)

For example, we might want our program to respond to any of the following:

prog -h
prog -H
prog -v
prog -f *filename*

To handle command line options, we need a means of distinguishing between parameters that are filenames, and parameters that are flags.

To use **getopts**, first establish the various flags the program is to understand. For example, for the above syntax, the options are **hHvf:**. The colon after the " f " indicates that the " f " is to be followed by an additional parameter (such as a filename).

For example:

```
79      :       while getopts "hHvlbf:" result
80      :       do
81      :          case $result in
           .
           .
           .
```

Each time the **while** loop runs, **getopts** is invoked, scans the parameters to the script, and places the first new option it finds in a special variable called **result**. The index number of the next shell argument to process is placed in another special variable called **OPTIND**, and if the flag has an optional argument (like the **f:** option above) the argument is placed in **OPTARG**. If **getopts** cannot find an option, it exits with a non-zero (or failure) exit value.

It is up to the shell script to retrieve all the options from a parameter list. So **optargs** is usually used in a structure called a **while** loop, explained below.

Repeating commands zero or more times: the while loop

A **while** loop differs from a **for** loop in that a **for** loop is executed a set number of times (for each item in its list), but a **while** loop is repeated indefinitely, or until some condition ceases to be true. The general format of a **while** loop is as follows:

> while *condition*
> do
>   *command*
>   .
>   .
>   .
> done

The *condition* is a command or test of some kind. (For an explanation of tests, see "Different kinds of test" (page 277).) If it exits with an exit value of 0, implying success, the commands in the body of the **do** loop are carried out; if it failed (has a non-zero exit value) the loop is skipped and the script continues to the next line.

Note that there is no guarantee that the commands in the body of the loop will ever be carried out. For example:

```
while [ "yes" = "no" ]
do
 some_command
    .
    .
    .
done
```

some_command will never be carried out, because the test [**"yes" = "no"**] always fails. On the other hand, the opposite effect can occur:

```
while [ "yes" ]
do
 some_command
    .
    .
    .
done
```

Because the literal string "yes" exists, **test** returns *true* all the time, so the loop repeats endlessly.

Repeating commands one or more times: the until loop

It is sometimes necessary to execute the body of a loop at least once. Although the **while** loop provides the basic looping capability, it does not guarantee that the body of the loop will ever be executed because the initial test may fail. For example, the body of the loop in the example above will never be executed because the test condition is always false.

We could make sure that the body of the loop was executed at least once by duplicating it before the **while** statement, like this:

```
some_command
while [ "red"="blue" ]
do
 some_command
done
```

However, this is prone to error when the loop body contains a lot of commands. Luckily the shell gives us a different type of looping construct: the **until** loop. An **until** loop looks very similar to a **while** loop; the difference is that the loop body is repeated *until* the test condition becomes false, rather than *while* it remains true.

For example, this loop will repeat infinitely, because the test always returns a non-zero (false) value:

```
until [ "red"="blue" ]
do
    some_command
done
```

By carefully choosing our test, we can ensure that the body of an **until** loop will be executed at least once: to do so, we must make sure that the test parameter is false. For example:

```
leave_loop="NO"
until [ leave_loop="YES" ]
do
    some_command
    .
    .
    .
    leave_loop="YES"
done
```

The body of this loop will be executed at least once. If we change the **until** on the second line to a **while**, the loop will never be entered.

Making choices and testing input

To handle the command line options to our script, lines 79 to 93 run **getopts** in a **while** loop. As long as **getopts** continues to return an option, the body of the loop is executed: when **getopts** can no longer detect any options, the **while** loop fails. **shift** is then used to discard the options.

Embedded in the loop to get options, we see another kind of statement: a **case** statement. Immediately after it, on lines 81 to 83, we see an **if** statement. These are both mechanisms for choosing between two or more options. **if** depends on the return value of a test condition; **case** operates by matching patterns.

When we need to repeat an operation a variable number of times, we must check after each repetition to determine whether it has produced the desired result. If not, we may need to repeat the task again: otherwise, we may want to do something else. The **if** statement allows us to choose between alternative courses of actions; the **test** or [...] command allows us to check whether a condition holds true. (The **case** statement can be used as a generalized form of **if** statement, for choosing between many options. We will deal with it later.)

Choosing one of two options: the if statement

The simplest form of **if** statement is illustrated on lines 81 to 84:

```
94   :     if [ $help = 'yes' ]
95   :     then
96   :        _help
97   :         exit 1
98   :     fi
```

The statement following **if** is evaluated. If it is true (that is, if it returns a value of 0), the body of the **if** statement (from **then** to **fi**) is carried out. If it is nonzero, the body of the **if** statement is skipped.

if has the following structure:

> if *condition*
> then
> > commands executed if *condition* succeeds
>
> fi

An alternative structure is the following:

> if *condition*
> then
> > commands executed if *condition* succeeds
>
> else
> > commands executed if *condition* does
> > not succeed
>
> fi

The following structure is also valid:

> if *condition1*
> then
> > commands executed if *condition1* succeeds
>
> elif *condition2*
> then
> > commands executed if *condition2* succeeds
>
> fi

condition is a command that returns an exit value: zero if successful or some other value if it failed. The **if** command carries out *test*, then executes the series of commands (from **then** to **else** or **fi**) if and only if *test* returned a value of "0" or **TRUE**. (**fi** is the command denoting the end of an **if** construct.)

If the **if** command contains an **else** portion, the commands between **else** and **fi** are only carried out if the test returns a result *other* than **TRUE**; that is, if the test statement fails.

If the **if** statement is followed by an **elif**, the **elif** statement is carried out if the condition tested by the previous **if** statement fails. An **elif** statement is otherwise identical with an **if** statement.

The following two lines of code have the same effect:

if [$answer = 'y']

if test $answer = 'y'

If the test succeeds, indicating that the value of **answer** is "y", then the first set of commands is carried out. Otherwise, the **else ... fi** section of the script is executed.

Different kinds of test

In general, tests are carried out either by enclosing them in square braces (as above) or by using the command **test**(C). The most useful tests are as follows:

-r *file*	True if a file called *file* exists and is readable.
-w *file*	True if a file called *file* exists and is writable.
-x *file*	True if a file called *file* exists and is executable.
-s *file*	True is a file called *file* exists and is not empty.
-d *file*	True if a file called *file* exists and is a directory.
-f *file*	True if a file called *file* exists and is a regular file.
-z *string*	True if the length of *string* is zero.
-n *string*	True if the length of *string* is non-zero.
string1 = *string2*	True if *string1* equals *string2*.
string1 != *string2*	True if *string1* is not equal to *string2*.
number1 **-eq** *number2*	True if the integer *number1* equals *number2*.
number1 **-ne** *number2*	True if the integer *number1* is not equal to *number2*.
number1 **-gt** *number2*	True if the integer *number1* is greater than *number2*.
number1 **-lt** *number2*	True if the integer *number1* is less than *number2*.
number1 **-ge** *number2*	True if the integer *number1* is greater than or equal to *number2*.
number1 **-le** *number2*	True if the integer *number1* is less than or equal to *number2*.

In addition to these tests, there are a number of others; see **test**(C) for details. In general, the tests listed here should be sufficient to let you test for the existence of files, to check whether your script has permission to manipulate a given file, to compare two numbers, and to see if a string matches some value. These are the commonest comparisons used to help a script decide on a course of action to take.

Testing exit values

In addition to the explicit **test** or [commands, **if** can make a choice on the basis of any program (or pipeline of programs) that returns a value. It is normal for programs to return "0" if they succeed, or another (usually negative) number if they fail; this value is retained in the variable $?, which is implicitly tested by **if**. It is not uncommon to see a shell script that contains commands like the following:

```
if who | grep -e "$1" > /dev/null
then
    print -- $1 is logged in
fi
```

In this example, the output from **who** is piped to **grep**. The **if** statement tests the output from the pipe, which is the value returned by **grep**. **grep** returns 0 if it finds the target string, or a non-zero value if it fails.

This example is therefore equivalent to a test that returns **TRUE** if a string is present in a given file.

The && and || operators

There are two compact versions of the **if** test which you may see from time to time; these tests operate on a single statement and determine whether a subsequent command is to be executed. They are && (AND IF) and || (OR IF). These operators evaluate $? for the previous command. && executes the following command if the previous command succeeded; || executes the following command if the previous command failed. Note that the execution of the second command is entirely dependent on the result of executing the first command. Thus, if you write a line with two or more of these operators, each command is executed in turn along the line until one of them results in a test failing.

For example, the test to see if a given user is logged on could be written as follows:

who | grep -e "$1" | | echo "$1 is not logged on"

A **who** listing is piped to **grep**, which searches for the subject (whose name is the first argument to the script). The OR IF test examines the returned value from **grep**. If **grep** failed (that is, if the user is not logged on), a message is printed. If **grep** succeeded and returned "0", no message is needed because **grep** printed the line from the **who** listing.

In general, you can use | | to execute a command when the previous command has failed, and you can use **&&** to execute a command if the previous command has succeeded.

For example, take the command:

compress $1 | | print "Something went wrong compressing $1"

The program **compress** is executed. When it finishes, its exit value $? is tested by | |. If it is non-zero, the error message is printed.

This compares with the other command:

compress $1 && print "Finished compressing $1"

If the exit value of **compress** is 0, the message is printed.

A common problem when using the **&&** and | | operators is to assume that they are equivalent to the logical operators provided by other programming languages. In fact, these operators are conditional constructs that evaluate strictly from left to right. Consequently it is hazardous to use them for evaluating logically true or false values (like the **&&** or | | operators in C). These operators are not strictly equivalent to **if ... else ... fi** either. For example, the following short script determines if someone is logged in:

```
if who | grep $1 >/dev/null
then
   echo $1 is logged in
else
   echo $1 is not logged in
fi
```

Using the && and | | operators, we might be tempted to rewrite this script more succinctly as follows:

```
who | grep $1 >/dev/null && echo $1 is logged in || echo $1 is not logged in
```

However, this version will execute the second **echo** incorrectly if the pipe (**who | grep $1**) fails. The **if ... else ... fi** version, in contrast, does not exhibit this behavior (despite looking superficially similar in logical terms).

Making multiway choices: the case statement

In explaining the large example program, we have so far ignored lines 82 to 91 These contain a structure designed to choose between several different options: a **case** statement.

```
81 :         case $result in
82 :              h|H)    help=yes              ;;
83 :              v)      verbose=yes           ;;
84 :              l)      record=yes
85 :                      log=off
86 :                      LOG=ON                ;;
87 :              b)      batch=yes             ;;
88 :              f)      file=yes
89 :                      fname=$OPTARG         ;;
90 :              *)      help=yes              ;;
91 :         esac
```

The **case** command is followed by a variable. This is tested against each of the options in turn, until the **esac** statement (signifying end of **case**) is reached.

In addition to setting variables, you can use branches of a **case** construct to call functions or exit. (An **exit** statement is used to exit from the current script.)

case statements are not essential to writing scripts that can handle multiway choices, but they make things easier. Consider the following alternative:

```
if [ ${result} = "h" ]
then
  help=TRUE
else
  if [ ${result} = "H" ]
  then
    help=TRUE
  else
    if [ ${result} = "v" ]
    then
      verbose=TRUE
    else
      if [ ${result} = "l" ]
      then
        record=TRUE
        log=off
        LOG=ON
      else
        if [ ${result} = "b" ]
        then
          batch="yes"
        else
          if [ ${result} = "f" ]
          then
            file=TRUE
            fname=${OPTARG:-unset}
          else
            help=TRUE;;
          fi
        fi
      fi
    fi
  fi
fi
```

This compound **if** statement does exactly the same thing as the earlier case statement, but is much harder to read and debug.

The general format of a **case** construct is as follows:

```
case $choice in
        1)      # carry out action associated with selection 1
        .

        .

        .

        ;;
        2)      # carry out action associated with selection 2
        .

        .

        .

        ;;
        3)      # carry out action associated with selection 3
        .

        .

        .

        ;;
        4)      # carry out action associated with selection 4
        .

        .

        .

        *)      # carry out action associated with any other selection
        .

        .

        .

        ;;
    esac
```

The **case** command evaluates its argument, then selects the matching option from the list and executes the commands between the closing parenthesis following the option and the next double semicolon. In this way, only one out of several possible courses of action can be taken. **case** tests the argument against its options in order, from top to bottom, and once it has executed the commands associated with an option it skips all the subsequent possibilities and the script continues running on the line after the **esac** command.

To trap any possible selection use an option like:

```
    *)      # match any possible argument to case
    .

    .

    .
    ;;
```

The * option matches any possible argument to the **case** construct; if no prior option has matched the argument, the commands associated with the * option are automatically carried out. For this reason, the *) option should be placed at the bottom of the **case** construct; if you place it at the top of the construct, the * option will always be executed before the shell has a chance to check any other options.

There is no effective size limit to a **case** construct, and unlike an **if ... then ... elseif** cascade the construct is "flat"; that is, it is an indivisible structure, and there is consequently no difficulty in working out which construct is being evaluated.

Generating a simple menu: the select statement

Although not used in the readability analysis sample program, the **select** statement can be used to simply generate menus. It is restricted to the Korn shell, and has no equivalent in the Bourne and C shells. It has the following syntax:

> **select** *name* [**in** *list*]
> **do**
>> *statements*   # statements use $name
> **done**

The **in** *list* construct can be omitted, in which case, *list* defaults to $@ (see "Passing arguments to a shell script" (page 250)).

The **select** statement generates a menu from the entries in *list*, one per line, with each preceded by a number. It also displays a prompt, by default a hash sign followed by a question mark (#?). The user's response to the prompt is stored in the variable **name**; on the basis of the value of *$name*, the appropriate statement is executed. **select** then prompts for another choice, unless an explicit **break** command causes the loop to terminate.

The following trivial sample code illustrates **select** in use:

```
print "Choose a dinosaur:"
select dino in allosaurus tyrannosaurus brontosaurus triceratops
do
case $dino in
allosaurus ) print "Jurassic carnosaur" ;;
tyrannosaurus ) print "Cretaceous carnosaur" ;;
brontosaurus ) print "Jurassic herbivore" ;;
triceratops ) print "Cretaceous carnosaur" ;;
*) print "invalid choice" ;;
esac
break
done
```

The following shows the code in use (the program is called *dino_db*):

```
Choose a dinosaur:
1) allosaurus
2) tyrannosaurus
3) brontosaurus
4) triceratops
#?
```

Expanding the example: counting words

At present, our readability analyzer program does very little processing. It can trap signals (preventing it from terminating if interrupted), it can scan the command line for arguments, and it sets up some useful routines for printing help and clearing the screen. However, we now want the program to perform a useful task.

Given the size of the skeleton structure we have already created, it might look as if it will take a lot of work to make it do anything useful. However, surprisingly little additional programming is needed.

As a first step towards writing a style analysis program, it would be useful to know how many words, characters and lines there are in the target file. We can use **wc** to obtain this information for any given file; we can also use backquotes to capture the output and process it.

To add word counting to our program, all we need to do is change the following lines:

```
23 :
24 :    do_something()
25 :    {
26 :      wordcount=`wc -w ${fname} | awk '{ print $1 }'`
27 :      lines=`wc -l ${fname} | awk '{ print $1 }'`
28 :      chars=`wc -c ${fname} | awk '{ print $1 }'`
29 :      echo "File ${fname} contains:
30 :            ${wordcount}\t\twords
31 :            ${lines}\t\tlines
32 :            ${chars}\t\tcharacters "
33 :    }
```

The main task of the program is to call the function **do_something**. This function runs **wc**, pipes the output through a short **awk** command, and traps the result in a variable; then it prints a formatted report.

For example:

```
$ rap -f rap

File rap contains:
        243             words
        95              lines
        1768            characters

$
```

The **awk** program '{ print $1 }' prints the first field on every line **awk** reads from the standard input. This is a typical **awk** program: short, integrated into a shell script, and used to carry out a transformation on a stream of text. For more information on using **awk**, see Chapter 13, "Using awk" (page 323).

The important point to note here is that by encapsulating the functionality of the program in a subroutine (the function **do_something**) we have made it a lot easier to change the program. (Ideally **do_something** would be written as three separate functions, to count words, lines, and characters. However, because it is comparatively short it is presented here as a single unit.)

We can make our program do something else entirely, simply by modifying **do_something** and changing the help text in **_help**. Most of the program is actually a skeleton that we can use to hang useful subroutines off: you can reuse it as a starting point for your own batch mode shell scripts.

Making menus

Starting from our current example, it is not difficult to turn the script into a fully interactive program with menus. We have already seen most of the structures we need: all that is necessary is to put them together in a different order.

The general structure of a batch mode script is as follows:

```
Define constants (variables that will not change)
Define functions (routines to handle specific jobs)
Set traps
get command line options with getopts
  use options to set control variables
for all in $*
do
  some_function()
done
```

The only element of repetition is the loop at the end, which repeats for each file passed to the script as an argument.

A menu driven script behaves differently:

```
Initialize variables and define functions
Repeat (until some ''exit'' state is reached)
{
  Display a menu
  Get the user's choice
  Do something with the choice (change state or call function)
}
On ''exit'' close files and quit
```

This process, an endless loop, is called a mainloop. The menu is displayed, then a function like **getc** (described in "Reading a single character from a file or a terminal" (page 260)) is used to retrieve a single keystroke. Such a function may either grab the first key the user presses, or let them correct the entry and press ⟨Enter⟩ before accepting input. (There are arguments for and against both strategies. In general, you should always give your users an opportunity to check their input, and correct any mistakes they may have made.)

Depending on the value of the key, an option is selected from a **case** statement. Each option either sets a variable, or calls a function (called a callback) which does something in the background, "behind" the menu. Finally, if the option to quit is selected, the **break** statement is executed to quit the loop.

Here is part of a menu based script, containing the mainloop:

```
282 : done
283 : if [ $help = "yes" ]
284 : then
285 :   _help
286 :   exit 1
287 : fi
288 : if [ $batch = "yes" ]
289 : then
290 :   analyze
291 :   exit 0
292 : fi
293 : #
294 : #---------- enter the mainloop -----------------------
295 : #
296 : while :
297 : do
298 :         echo $CLS
299 :         echo "
300 :
301 :     ${HILITE}Readability Analysis Program${NORMAL}
302 :
303 :     Type the letter corresponding to your current task:
304 :
305 :     f         Select files to analyze [now ${HILITE}$fname${NORMAL} ]
306 :     p         Perform analyses
307 :     l         switch ${next_log_state} report logging [now ${HILITE}$log${NORMAL}]
308 :     q         quit program
309 :
310 :
311 :     =======>"
312 :         getc char
313 :         case $char in
314 :         'f') getloop=1
315 :             get_file ;;
316 :         'p') analyze
317 :                     strike_any_key      ;;
318 :         'l') toggle_logging    ;;
319 :         'q') break            ;;
320 :         *)   continue ;;
321 :         esac
322 : done
323 : clear
324 : exit 0
```

The first part of this extract, lines 283 to 292, check to see whether help is to be printed, or the script is to be run in batch mode: if the answer to the latter question is yes, a function called **analyze** is called and the script exits without presenting a menu. Then we see the mainloop, from line 284 to 324. **$endloop** is initially set to NO, so the test at the top of the loop evaluates to true: therefore the body of the **do** loop is executed at least once.

Within the loop, a menu is printed and then the script waits for the user to press a key. The character that is read is used to trigger a **case** statement (lines 312 to 321) that either modifies the state of some variables, or calls a function (like **analyze**, which does the analysis work, or **getfile**, which prompts the user for the name of a file to work on, or **strike_any_key**, which prints a message like "Press any key to continue").

Note the use of reverse video in the menu to emphasize important information. In general, you should try to make menu driven interfaces guide the user through to the next step in an intuitive and natural manner. One way of doing this is to highlight the important default information (like the file to be processed), in close proximity to the option that changes it (like the option to select a file to analyze).

Also worth noting is the use of "toggle" variables, that switch an additional feature on or off. The variables **$log** and **$next_log_state** perform this function for logging. They are switched within a separate function, **toggle_logging**:

```
83 : toggle_logging ()
84 : {
85 :    log=$next_log_state
86 :    case $log in
87 :      ON)  next_log_state=OFF ;;
88 :      OFF) next_log_state=ON  ;;
89 :    esac
90 : }
```

log indicates whether output is to be logged to a file; **next_log_state** is used in a message display that tells the user whether they can switch logging on or off. (By definition, **next_log_state** and **log** must be in opposite states at all times.)

It is very easy for a mainloop to become too big to read. For this reason, any task that has more than one step is farmed out to another function. This includes the display of submenus. For example, **get_file** uses a menu to select a file to check:

```
145 : get_file()
146 : {
147 :   while :
148 :   do
149 :      echo $CLS
150 :      echo "
151 :
152 :      ${HILITE}Select a file${NORMAL}
153 :
154 :      Current file is: [${HILITE} $fname ${NORMAL}]
155 :
156 :      Type the letter corresponding to your current task:
157 :
158 :      [space]   Enter a filename or pattern to use
159 :      l         List the current directory
160 :      c         Change current directory
161 :      q         quit back to main menu
162 :
163 :
164 :      =======>"
165 :      getc char
166 :      case $char in
167 :      ' ')     get_fname            ;;
168 :      'l')     ls | ${PAGER:-more} ;;
169 :      'c')     change_dir                  ;;
170 :      'q')     break              ;;
171 :      *)                          ;;
172 :      esac
173 :      strike_any_key
174 :   done
175 : }
```

This function contains a couple of features that do not appear in the main-loop. Notably, it calls a routine for changing directory, a routine for getting a filename, and lists the contents of a directory (using the pager indicated by the environment variable **PAGER**, or **more** if **PAGER** is not set).

Assigning variables default values

Line 168 shows an example of providing a default value for a variable. We have already seen how to assign a value to a variable. For example:

value=$newvalue

This assigns the value of **newvalue** to **$value**. But there are times when we want to provide a default option, in case **$newvalue** is bogus (for example, if the user accidentally pressed ⟨Enter⟩ instead of entering a name). An assignment of the form **variable=${value:-***default***}** assigns **value** to **$variable** if it is set: otherwise it assigns *default* to **$variable**. In the example above, the variable **${PAGER:-more}** is expanded to either the value of **$PAGER**, or if this is not set, to **more**.

For example, here is **get_fname**:

```
94 : get_fname ( )
95 : {
96 :    echo "Enter a filename: \c"
97 :    read newfname
98 :    fname=${newfname:-${fname}}
99 : }
```

At the beginning of the script (we have not yet looked at this in detail) **fname** is set to " " (a space character). So if the user fails to enter a reasonable value, it remains " ".

There are other uses for this mechanism. For example:

```
117 :    newdir=${newdir:-`pwd`}
```

This line sets **newdir** (the directory to change to) to the newly entered directory, or (if nothing is specified) to the current working directory.

Variations exist on the default behavior for a variable assignment. Some of the most common variable substitutions you can use are as follows:

${*var*:-*word*}   If *var* is set and not empty, substitute the value of *var*; otherwise substitute *word*.

${*var*:=*word*}   If *var* is not set or is empty, set it to *word*; then substitute the value of *var* (that is, if **$var** does not exist, set **$var** to **$word** and use that).

${*var*:?*word*}   If *var* is set and not empty, substitute the value of *var*; otherwise print *word* and exit from the shell.

${*var*:+*word*}   If *var* is set and not empty, substitute the value of *word*; otherwise substitute nothing.

The Korn shell provides additional substitutions for matching patterns and substituting the size of variables: see **ksh**(C) for details.

Tuning script performance

The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum cost in terms of user time. This entails reducing both the effort that the user has to put into achieving their goal, and the time taken.

Good shell programming technique relies on an understanding of the desired goal and the ability to write clear, easily debugged scripts, but you can also add efficiency through awareness of a few simple rules of thumb.

An effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources. Your time, as the programmer, is almost certainly more expensive than the computer's.

How programs perform

A general law of programming, proven through long experience, is that in any program the computer spends 90% of its time processing about 10% of the code. A second general law is that as programs age and are maintained, the changes introduced to them tend to add complexity to the original structure and reduce their efficiency. In this section, we'll look at program performance and means of improving it.

The flow of control within a program is determined by two types of construct; the loop construct and the branch construct. In batch programs such as filters, these are used in conjunction so that the program does something like this:

```
# generic filter program
#
read command line arguments
using getopts, for each flag {
        set a variable
        }
open input and output files
while (input != FALSE) {
        read in some data
        do something with it
        write it to the output file
        if an error occurred, exit with a message
        }
close input and output files
exit
```

The first action taken by this generic program is to check its command line for flags. Using a loop, it reads through each argument in turn and sets up any internal variables it needs. This loop is only used by the program when it starts up; for this reason it is called *initialization* code.

Having "parsed" its arguments, the program now opens its data files. An input and an output file are the lowest common denominator; some programs open several files each for input and output, but this is a simple, generic example. Again, opening the files is only carried out once. Note that in a real program each attempt to open a file will be enclosed in an **if** construct that checks for errors; if the attempt fails, the **else** part of the **if** construct usually causes the program to exit with an error message.

The program now enters a loop, reading data from the input file, doing something to it, and writing it to the output file, while the input is available. (By convention, if an operation succeeds it usually returns a value of 0.) This is the meat of the program; it is where the activity for which the program was written takes place, and it is repeated for a number of times proportional to the amount of data in the input files.

When the program can no longer read any more input, it exits the main loop and executes the termination code of the program. Termination code is used to tidy up after the main loop; to close open files and write a final message to the output. (The command **wc**, which counts words, uses its termination code to print out a final sum of all the words it counted in its main loop.) This section of the program, like the initialization code, is only executed once.

This program structure is not universal, but it is sufficiently common to be worth using as a model to demonstrate how to tune your programs, and it accounts for the vast majority of shell scripts and non-interactive filters. While shell scripts rarely open data files and process them directly, they frequently invoke other programs which do just that; consequently, the same general techniques for improving performance are applicable to them.

How to control program performance

As mentioned earlier, in any shell script, 90% of the computational load is imposed by about 10% of the script. The bottlenecks to look out for are as follows:

- Loops, especially the main program loop. A process which is called repeatedly imposes a heavy load on the computer. Most shell script loops are extremely heavy users of computer resources because they exec programs several times in rapid succession.

- File access (and reads and writes directed through named pipes). Because the computer's hard disk is several orders of magnitude slower than its memory, any procedure that involves heavy disk I/O will invariably impose a heavy load on the system.

- Processes. Many commands are built into the shell; but those which are not require the system to load and execute a program. This has two consequences; a disk access is required, and an additional process is run (diverting resources from any other processes which are being executed concurrently).

- Size of data. It should be obvious that as the files that are being processed by a filter grow longer, all processes involving the file take longer. However, the relationship between file size and time is not fixed; big files may take much longer to process than several small files containing the same total amount of information.

To improve the performance of a shell script, you need to be constantly aware of these considerations. Any activity that takes place in a main loop is likely to yield a big performance improvement if you can find a way to reduce the amount of disk I/O or number of processes it requires. Activities that require a large data file may be speeded up by switching to several smaller files, if possible. (A small file is one that is less than eight or ten kilobytes long; for technical reasons such files can be opened and scanned more rapidly than larger files.)

The standard development cycle, which should be applied to shell procedures as to other programs, is to write code, get it working, thoroughly test it, measure it, and optimize the important parts (outlined above), looping back to earlier stages wherever necessary. The **time**(C) command is a useful tool for optimizing shell scripts. **time** is used to establish how long a command took to execute:

```
$ time ls
real    0m0.06s
user    0m0.03s
sys     0m0.03s
```

The values reported by **time** are the elapsed time during the command (the real time); the time the system took to execute the system calls within the command (the "sys" time); and the time spent processing the command itself (the user time). In practice, only the first value, the real time, is relevant at this level. Note that this is the output from the Korn shell's built-in **time** command; the Bourne shell output may vary. (If you have the Development System, the **timex**(ADM) command offers additional facilities.)

Because the SCO OpenServer system is multi-tasking, it is impossible to accurately judge how long a program is taking to run by any other means; a seemingly slow process may be the result of an unusually heavy load being placed on the computer by some other user or process. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

A useful technique is to encapsulate the body of a loop within a function, so that the sole activity within the loop is to call that function; you can then **time** the function, and time the loop as a whole. Alternatively, you can time individual steps in the process to see which of them are taking longest.

Number of processes generated

When you execute large numbers of short commands, the actual execution time of the commands might be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is an alphabetical list of those that are built in to the Korn shell and Bourne shell (**select** is Korn shell only):

break	case	cd	continue	echo
eval	exec	exit	export	for
if	read	readonly	return	select
set	shift	test	times	trap
umask	until	wait	while	.
:	{}			

Note that **echo** and **test** also exist as external programs. Some other external commands have been added to the shells, but they are nonstandard and their use will impact the performance of shell scripts on other systems.

Parentheses, ( ), are built into the shell, but commands enclosed within them are executed as a child process; that is, the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**. The disadvantage of this is that when another process is **exec**ed it is necessary to perform a disk I/O request to load the new program. Even if the program is already in the buffer cache (an area of memory used by the system to store frequently accessed parts of the filesystem for rapid retrieval) this will increase the overhead of the shell script.

You should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by the following:

processes = (k*n) + c

where k and c are constants for any given script, and n can be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of k, sometimes to zero. Any procedure whose complexity measure includes n^2 terms or higher powers of n is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named *file2lower*, whose text is as follows:

```ksh
#!/bin/ksh
#
# file2lower -- renames files in parameter list to
#               all-lowercase names if appropriate
#
PATH=/bin:/usr/bin
for oldname in "$@"
do
        newname=`echo $oldname | tr "[A-Z]" "[a-z]"`
        if [ $newname != $oldname ]
        then
        {
                if [ ! -d "$oldname" ]
                then
                {
                        mv "$oldname" "$newname"
                        print "Renamed $oldname to $newname"
                }
                else
                        print "Error: $oldname is a directory" >&2
                fi
        }
        fi
done
```

This shell script checks all the names in its parameter list; if a file of that name exists, is writable, and contains uppercase letters in its name, it is renamed to a lowercase equivalent. This is useful when copying files from a DOS filesystem, because files imported from DOS have all uppercase names.

For each iteration of the main **do** loop, there is at least one **if** statement. In the worst case, there are two **if**s, an **mv** and a **print**. However, only **mv** is not built into the shell. If n is the number of files named by the parameter list, the number of processes tends towards $(4*n)+0$. (The c term of the equation given above is applicable to commands executed once before and after the loop.)

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C or **awk**. Shell procedures should not be used to scan or build files a character at a time.

Number of data bytes accessed

It is worth considering any action that reduces the number of bytes read or written. This might be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to **sort** will be much smaller:

```
sort file | grep pattern
grep pattern file | sort
```

Shortening data files

There are two good reasons for using short files (less than 10,000 bytes, if possible; certainly less than a quarter of a megabyte). Firstly, the traditional UNIX filesystems access short files faster than long files. Significant overheads are incurred in reading or writing to a file that is, in the first instance, more than 10KB long, and in the second instance, more than 256KB long (or, in an extreme case, more than 64MB long). With each successive increase in size, the process of reading from or writing to the file becomes slower; therefore short files are preferred.

In addition, the performance of some programs degrades significantly as their input files increase in size. Any complex sorting or comparison operation (using **sort** or **diff**) usually takes significantly longer to perform on a single large file than on two smaller files containing the same amount of information. This degradation is an unavoidable consequence of the nature of the problem these programs are dealing with and can rarely be worked around, although it is not significant when working with short files.

Shortening directory searches

Directory searching consumes a lot of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd**, the *change directory* command, can help shorten long pathnames and thus reduce the number of directory searches needed. For example, try the following commands:

time ls -l /usr/bin/* >/dev/null
time cd /usr/bin; ls -l * >/dev/null

The second command runs faster because of the fewer directory searches.

Directory-search order and the PATH variable

The **PATH** variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result might be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current **PATH** variable. As an example, consider the effect of invoking **nroff** (that is, */usr/bin/nroff*) when the value of **PATH** is *:/bin:/usr/bin*. The sequence of directories read is as follows:

```
.
/
/bin
/
/usr
/usr/bin
```

A long path list assigned to **PATH** can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and in */usr/bin*. Careless **PATH** setup can lead to unnecessary searching. The following three examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/local/bin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/local/bin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/local/bin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands can often be speeded up by setting the **PATH** variable inside the procedure so that the fewest possible directories are searched in an optimum order.

Recommended ways to set up directories

It is wise to avoid directories that are larger than necessary, for the same reason that you should avoid large files; directories are a special type of file, and when a directory grows too large any process that searches it becomes slower.

You should be aware of several special sizes. A directory that contains entries for up to 62 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. A directory can have up to 638 entries and still be viable, as long as it is used only for data storage; anything larger is usually a disaster when used as a working directory. The figures 62 and 638 apply to filenames of 14 characters or less. As filename lengths increase, up to a maximum of 255 characters, the number of files that fit on a single disk block decreases, thus reducing the optimum number of files in a directory.

It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 62 or 638 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

Putting everything together

We have covered most of the shell-specific elements of a style analysis program, except for two components: the global constants set up at the top of the file, and the function **analyze**, which reports on the readability indices of a file. Here is a complete listing of the program. (See below for a commentary on the features that have not yet been covered.)

```
1  : #--------------------------------------------------------
2  : #
3  : # rap -- Readability Analysis Program
4  : #
5  : # Purpose: provide readability analysis of texts to:
6  : # Kincaid formula, ARI, Coleman-Liau Formula, Flesch
7  : # Reading Ease Score. Also word count, sentence length,
8  : # word length.
9  : #
10 : # Note that rap is _not_ as functional as style(CT),
11 : # which is dictionary-driven; this is the outcome of
12 : # a deliberate attempt to keep everything in a single
13 : # shell script.
14 : #
15 : #------------- define program constants here ----------
16 : #
17 : DEBUG=${DEBUG:-true}
```

```
18 : CLS=`tput clear`
19 : HILITE=`tput smso`
20 : NORMAL=`tput rmso`
21 : #
22 : #----- define the lexical structure of a sentence -----
23 : #
24 : # a `word' primitive is any sequence of characters.
25 : #
26 : WORD='[A-Za-z1-90]+'
27 : #
28 : # whitespace is what goes between real words in a sentence;
29 : # it includes carriage returns so sentences can cross line
30 : # boundaries.
31 : #
32 : WHITESPACE="[[:space:]]"
33 : #
34 : # an initial -- one or two letters followed by a period --
35 : # is defined so we call tell that it is not a short sentence.
36 : # (Otherwise Ph.D. would be counted as two sentences.)
37 : #
38 : INITIAL="($WHITESPACE|.)(([A-Za-z0-9]|[A-Za-z0-9][A-Za-z0-9]).)"
39 : #
40 : # syllabic consonants; consonants including letter pairs:
41 : #
42 : CONS="[bcdfghjklmnpqrstvwxyz]|ll|ght|qu|([wstgpc]h)|sch"
43 : #
44 : # syllabic vowels; include the ly suffix
45 : #
46 : VOWL="[aeiou]+|ly"
47 : #
48 : # definition of a syllable (after Webster's Collegiate Dictionary)
49 : #
50 : SYL="(${CONS})*\
51 : ((${CONS})|((${VOWL})+))\
52 : (${CONS})*"
53 : #
54 : # Finally, a sentence consists of (optionally) repeated
55 : # sequences of one word followed by zero or more
56 : # whitespaces, terminated by a period.
57 : #
58 : SENT="($WORD($WHITESPACE)*)+."
59 : #
60 : #---------- initialize some local variables -----------
61 : #
62 : SCRIPT=$0
63 : help='no'   ;    verbose=' '    ;    record=' '
64 : next_log_state='ON';    log='OFF'    ;    batch=' '
65 : file=' '    ;    fname=' '    ;    LOGFILE=$$.log
66 : #
67 : #-------------- define program traps here ------------
```

```
68 : #
69 : trap "strike_any_key" 1 2 3 15
70 : #
71 : #---------------- useful subroutines -----------------
72 : #
73 : getc ()
74 : {
75 :   stty raw
76 :   tmp=`dd bs=1 count=1 2>/dev/null`
77 :   eval $1='$tmp'
78 :   stty cooked
79 : }
80 : #
81 : #-----------------------------------------------------
82 : #
83 : toggle_logging ()
84 : {
85 :   log=$next_log_state
86 :   case $log in
87 :     ON)  next_log_state=OFF ;;
88 :     OFF) next_log_state=ON  ;;
89 :   esac
90 : }
91 : #
92 : #-----------------------------------------------------
93 : #
94 : get_fname ()
95 : {
96 :   echo "Enter a filename: \c"
97 :   read newfname
98 :   fname=${newfname:-${fname}}
99 : }
100 : #
101 : #-----------------------------------------------------
102 : #
103 : strike_any_key()
104 : {
105 :     echo '
106 :     strike any key to continue ...\c'
107 :     getc junk
108 :     echo $CLS
109 : }
110 : #
111 : #-----------------------------------------------------
112 : #
113 : change_dir ()
114 : {
115 :   echo "Enter a directory: \c"
116 :   read newdir
117 :   newdir=${newdir:-`pwd`}
```

```
118 :   cd $newdir
119 :   echo "Directory set to: $newdir"
120 : }
121 : #
122 : #--------------------------------------------------------
123 : #
124 : _help()
125 : {
126 : echo "
127 :
128 :   Readability Analysis Program
129 :
130 :   A shell/awk demo to determine the readability grade of texts
131 :
132 :   Usage:
133 :
134 :   Either invoke with no options for full menu-driven
135 :   activity, or use the following flags:
136 :
137 :   -[h|H]  prints this help
138 :   -l      cause output to be logged to a file
139 :   -f file enter the name of the file to check
140 :   -b      run in batch mode (no menus)
141 : "
142 : }
143 : #
144 : #---------- define the menu handler functions here ----
145 : get_file()
146 : {
147 :   while :
148 :   do
149 :     echo $CLS
150 :     echo "
151 :
152 :   ${HILITE}Select a file${NORMAL}
153 :
154 :   Current file is: [${HILITE} $fname ${NORMAL}]
155 :
156 :   Type the letter corresponding to your current task:
157 :
158 :           [space] Enter a filename or pattern to use
159 :           l       List the current directory
160 :   c       Change current directory
161 :           q       quit back to main menu
162 :
163 :
164 :     =======>\c"
165 :     getc char
166 :     case $char in
167 :     ' ')    get_fname             ;;
```

```
168 :     'l')    ls | ${PAGER:-more} ;;
169 :     'c')    change_dir        ;;
170 :     'q')    break             ;;
172 :     esac
173 :     strike_any_key
174 :   done
175 : }
176 : #
177 : #----------------------------------------------------
178 : #
179 : analyze()
180 : {
181 :   if [ $fname = " " ]
182 :   then
183 :       echo "
184 :
185 :             You must specify a filename first
186 : "
187 :       strike_any_key
188 :       return 1
189 :   fi
190 :   wordcount=`wc -w < $fname`
191 :   lines=`wc -l < $fname`
192 :   nonwhitespace=`sed -e "/${WHITESPACE}/s///g" < $fname | wc -l`
193 :   sentences=`awk -e ' BEGIN { sentences = 0
194 :                       target = ""
195 :                       marker = "+X+"
196 :                     }
197 :                     { target = target " " $0
198 :                       initials = gsub(init, "", target)
199 :                           hit = gsub(sent, marker, target)
200 :                           sentences += hit
201 :                       if (hit != 0) {
202 :                           for (i= 0; i < hit; i++) {
203 :                                   found = index(target, marker)
204 :                             target = substr(target, found+3)
205 :                           } # end for
206 :                       } # end if
207 :                       hit = 0
208 :                     }
209 :           END   { print sentences }
210 : ' sent="$SENT" init="$INITIAL" < $fname`
211 :   letters=`expr $nonwhitespace - $lines`
212 :   sylcount=`awk -e ' BEGIN { sylcount = 0 }
213 :                      { target = $0
214 :                   sylcount += gsub(syllable, "*", target)
215 :                     }
216 :           END   { print sylcount }
217 : ' syllable="$SYL" < $fname`
218 :     echo "
```

```
219 :
220 :    Number of words: $wordcount
221 :    Number of syllables: $sylcount
222 :    Number of sentences: $sentences
223 :
224 :    "
225 :    export letters wordcount sentences sylcount
226 :    ARI=`bc << %%
227 :    l = ($letters / $wordcount)
228 :    w = ($wordcount / $sentences)
229 :    4.71 * l +0.5 * w -21.43
230 : %%
231 :  ` `
232 :    Kincaid=`bc << %%
233 :    w = ($wordcount / $sentences)
234 :    s = ($sylcount / $wordcount)
235 :    11.8 * s + 0.39 * w - 15.59
236 : %%
237 :    `
238 :    CLF=`bc << %%
239 :    l = ($letters / $wordcount)
240 :    s = ($sentences / ($wordcount / 100))
241 :    5.89 * l - 0.3 * s - 15.8
242 : %%
243 :    `
244 :    Flesch=`bc << %%
245 :    w = ($wordcount / $sentences)
246 :    s = ($sylcount / $wordcount)
247 :    206.835 - 84.6 * s - 1.015 * w
248 : %%
249 : `
250 :    if [ log = "ON" ]
251 :    then
252 :    echo "
253 :      ARI = $ARI
254 :      Kincaid= $Kincaid
255 :      Coleman-Liau = $CLF
256 :      Flesch Reading Ease = $Flesch" > $LOGFILE
257 :    fi
258 :    echo "ARI = $ARI
259 :    Kincaid= $Kincaid
260 :    Coleman-Liau = $CLF
261 :    Flesch Reading Ease = $Flesch" > /dev/tty
262 : }
263 : #
264 : #=========== THIS IS WHERE THE PROGRAM BEGINS =========
265 : #
266 : #
267 : #---------- parse the command line--------------------
268 : #
```

```
269 : while getopts hHvlbf: result
270 : do
271 :   case $result in
272 :   h|H)    help="yes"                ;;
273 :   v)      verbose="yes"             ;;
274 :   l)      record="yes"
275 :               next_log_state=off
276 :               log=ON                ;;
277 :   b)      batch="yes"               ;;
278 :   f)      file="yes"
279 :               fname=${OPTARG:-" "}   ;;
280 :       *)    help="yes"              ;;
281 :   esac
282 : done
283 : if [ $help = "yes" ]
284 : then
285 :   _help
286 :   exit 1
287 : fi
288 : if [ $batch = "yes" ]
289 : then
290 :   analyze
291 :   exit 0
292 : fi
293 : #
294 : #---------- enter the mainloop -----------------------
295 : #
296 : while :
297 : do
298 :         echo $CLS
299 :         echo "
300 :
301 :   ${HILITE}Readability Analysis Program${NORMAL}
302 :
303 :   Type the letter corresponding to your current task:
304 :
305 :   f   Select files to analyze [now ${HILITE}$fname${NORMAL} ]
306 :   p   Perform analyses
307 :   l   switch ${next_log_state} report logging [now ${HILITE}$log${NORMAL}]
308 :   q   quit program
309 :
310 :
311 :   =======>\c"
312 :       getc char
313 :       case $char in
314 :   'f') getloop=1
315 :       get_file   ;;
316 :   'p') analyze
317 :       strike_any_key     ;;
318 :   'l') toggle_logging    ;;
```

```
319 :    'q') break           ; ;
320 :    (**) continue   ; ;
321 :    esac
322 : done
323 : clear
324 : exit 0
```

The variable definitions from lines 17 to 65 set up some constants for screen clearing and highlighting, initialize variables for use in the script, and define some extended regular expressions, as explained in Chapter 12, "Regular expressions" (page 315), that are used later to scan the target file for initials, sentences, and syllables. The mechanism used to conduct the scan is a pair of scripts written in the **awk** programming language (explained in Chapter 13, "Using awk" (page 323)) that identify the number of sentences in a file, and the number of syllables in the file. These scripts lie between lines 190 and 217; they are explained in detail in "Spanning multiple lines" (page 364).

Readability analysis

Four different readability statistics are calculated within **analyze**. Readability statistics assess variables including the average number of words per sentence, average length of sentences, number of syllables per word, and so on, to derive a formulaic estimate of the "readability" of the text. They do not take into account less quantifiable elements such as semantic content, grammatical correctness, or meaning. Thus, there is no guarantee that a text that a readability test identifies as easy to understand actually is readable. However, in practice it has been found that real documents that the tests identify as "easy to read" are likely to be easier to comprehend at a structural level.

The four test formulae used in the **analyze** function are as follows:

Automated Readability Index
> The Automated Readability Index (ARI) is based on text from grades 0 to 7, and intended for easy automation. ARI tends to produce scores that are higher than Kincaid and Coleman-Liau, but are lower than Flesch.

Kincaid formula
> The Kincaid formula is based on navy training manuals ranging from 5.5 to 16.3 in grade level. The score reported by the formula tends to be in the mid-range of the four formulae. Because it is based on adult training manuals rather than schoolbook text, this formula is most applicable to technical documents.

Coleman-Liau Formula
> The Coleman-Liau formula is based on text ranging from .4 to 16.3. This formula usually yields the lowest grade when applied to technical documents.

Flesch Reading Ease Score
  The Flesch formula is based on grade school text covering grades 3 to 12. The difficulty score is reported in the range 0 (very difficult) to 100 (very easy).

To calculate these metrics, **analyze** first counts the number of words, lines and sentences in the target file, generating output like the following:

```
File rap-bat.wc contains:
        243             words
        95              lines
        1768            characters
```

Sentences are counted using a custom **awk** script, explained in "Spanning multiple lines" (page 364). Then the number of letters is established (by subtracting the white space from the file and counting the number of characters), and the number of syllables is estimated using another **awk** script. Finally, these values are fed into four calculations that make use of **bc**, the SCO Open-Server binary calculator.

bc is a simple programming language for calculations; it recognizes a syntax similar to C or **awk**, and can use variables and functions. It is fully described in **bc**(C), and is used here because unlike the shell's **eval** command, it can handle floating point arithmetic (that is, numbers with a decimal point are not truncated). Because **bc** is interactive and reads commands from its standard input, the basic readability variables are substituted into a here-document which is fed to **bc**, and the output is captured in another environment variable. For example:

```
233 :   Flesch=`bc << %%
234 :   w = ($wordcount / $sentences)
235 :   s = ($sylcount / $wordcount)
236 :   206.835 - 84.6 * s - 1.015 * w
237 :   %%
238 :   `
```

analyze also prints the output from the tests, as follows:

```
ARI = -10.43
Kincaid= -7.01
Coleman-Liau = -17.00
Flesch Reading Ease = 184.505
```

Depending on the setting of **$LOG** (the variable that controls file logging) the output is printed to the terminal, or printed to the terminal and a logfile (the name of which is set by the variable **$LOGFILE**.)

Extending the example

The readability analysis program presented above is a useful starting platform for writing your own programs. It provides a skeleton that can be used for either a batch script or an interactive, menu-driven application. It traps unwanted signals and ignores them. It demonstrates how to call short programs written in other languages (**bc** and **awk**) from within the shell. Finally, it provides a basic mainloop with callback functions that can be added to.

If you want to customize the script for your own purposes, the place to start is in the callback functions. Strip out the existing functions, and replace them with your own: then change the here-document that displays the opening menu. If you change the keys that trigger the callback functions, remember to modify the **case** statement below the menu. You can add as many extra callbacks as you like to the menu, but it is a good idea not to provide too many options on any one screen: remember that your users can become confused if confronted with too many choices or too much information.

Other useful examples

This section gives examples of some other useful procedures for automating tasks. All the scripts and sections listed below are intended to run under the Korn shell; you may have to modify them if you want to use the Bourne shell.

Mail tools

The following tools are used for manipulating mail folders and sending large files through mail.

Count the number of messages in MMDF mail folder

Consider the following script:

```
cnt=`grep '^A^A^A^A' $1 |wc -l`
print $(( cntot = cnt / 2 ))
```

MMDF stores messages in a folder as continuous ASCII text, delimited at top and bottom by a line containing four ⟨Ctrl⟩**A** characters. This script searches for the message delimiters and sets **cnt** to the number of lines containing delimiters. It then uses the Korn shell arithmetic facility to divide this total by two (because there are twice as many delimiters as messages). Thus, this script prints the number of messages in a MMDF mail folder.

It is not appropriate to use this script on a XENIX-format mail folder.

> **NOTE** To enter the ⟨Ctrl⟩**A** characters in the script using **vi**, press ⟨Ctrl⟩**V** then ⟨Ctrl⟩**A** for each character.

Print the header lines of every message in a folder

The following short script searches the files named by its positional para-
meters for lines beginning with the string "Subject:".

```
grep "^Subject:" $*|cut -c9-7
```

Mail headers consist of a series of lines beginning with keywords, like this:

```
From:
To:
Subject:
Date:
Organization:
Sender:
Reply-To:
Message-Id:
X-Mailer:
Status:
```

The subject lines are printed through a pipe to **cut**, which chops out and prints
only character positions 9 through 71 on each line (thus removing the string
"Subject:" and truncating long lines).

Note that this script makes no allowances for mail messages that contain
other (quoted) messages without indentation. To do this, it would be neces-
sary to write a longer script. (Hint: The end of a mail message is indicated by
two lines containing four ⟨Ctrl⟩A characters each. Valid mail messages can
have only one "Subject:" line. A better script would search for the first occur-
rence of a "Subject" line following a sequence of "^A^A^A^A".) Note also that
the "Subject:" line is not mandatory, so this script will miss messages that lack
a subject line altogether.

Mail a large file, in pieces

Note that the line numbers in this example are not part of the script, but are
provided for clarity: script, but are provided for clarity:

```
 1 :    #! /bin/ksh
 2 :    #
 3 :    #----- blocksize*80 is the maximum size of each chunk created
 4 :    #
 5 :    blocksize=512
 6 :    #
 7 :    #----- perform sanity checks on input
 8 :    #
 9 :    case $# in
10 :    2)      : break
11 :            ;;
12 :    *)      echo "
13 :
14 :            $0 <user> <file>
```

```
15 :
16 :              compress, uuencode, split into 1000 line chunks and mail
17 :              <file> to <user>.
18 :
19 :              This script is used to send large files (greater than
20 :              32KB) via email. <user> must be a valid mail address;
21 :              On completion, chunk will send a status report to you
22 :              via email.
23 :              "
24 :
25 :              exit 2
26 :              ;;
27 :     esac
28 :     #
29 :     #--------- test for a valid file -----------
30 :     #
31 :     target=$2
32 :     user=$1
33 :
34 :     [ -s "$target" -a -r "$target" ] || {
35 :             print -- Missing, empty or not readable: $target >&2
36 :             exit 1
37 :     }
38 :     #
39 :     # -------- end of sanity checks ------------
40 :     #
41 :     tmpdir=${TMPDIR:-/u/tmp}/$$
42 :
43 :     mkdir $tmpdir || exit 1
44 :     compress < $target | uuencode $target | (cd $tmpdir; split -$blocksize)
45 :     cd $tmpdir
46 :     for chunk in *
47 :     do
48 :             mail -s "section $chunk of $target" $user < $chunk &&
49 :                     print "Sent section $chunk at"; date
50 :     done 2>&1 | mail -s "Result of sending $target" $user
51 :     cd
52 :     rm -rf $tmpdir
```

This script (called *chunk*) takes two arguments; a valid mail address and a filename. Because the consequences of proceeding on the basis of a bad argument list could be messy, some checks are carried out (from lines 9 to 27). The **case** statement on line 9 tests whether there are too few arguments, and aborts with a usage message if this is the case.

The real work of the script is carried out from lines 41 to 52: **target** has previously been assigned the name of the file to transmit. The file is compressed, and uuencoded, then piped through **split** into sequentially named chunks of **blocksize** lines that are stored in **$tmpdir**.

Some mail gateways will not handle messages which are more than some arbitrary size; therefore the exact size of the chunks created by this mailer is defined in a single variable which can be adjusted easily.

A **for** loop now iterates over each chunk and invokes **mail**. Because the chunks contain no human readable information, it is vital to incorporate the name of each chunk in the message header.

Finally, a record of the transmission is mailed to the recipient, so that they know what to do with the pieces.

To reassemble a file from its component pieces, save the pieces (in order) to a file, edit the file to remove mail headers and blank lines, uudecode the file, and uncompress it. This method can be used to send large files through size-restricted mail gateways.

File tools

The following scripts are used for manipulating and returning information on files.

Return the total size of a group of files

The following is a script called *filesize*:

```
l "$@" | awk ' {        s += $5
                        f = f" "$NF
            }
        END {   print s, "bytes in files:", f} '
```

The l command (equivalent to **ls -l**) returns a long listing, the fifth field of which contains the size of a file in bytes. This script obtains a long listing of each file in its argument list, and pipes it through a short **awk** script. For each line in its standard input, the script adds the fifth field of the line to the variable **s** and appends the last field (the filename) to the variable **f**; on reaching the end of the standard input, it prints **s** followed by a brief message and **f**.

Compress a batch of files concurrently

The **compress**(C) command can compress a batch of files listed as arguments; however, if you run **compress** in this way only one process is created, and it compresses each file consecutively.

The following code is a script called *squeeze*:

```
((jobcount=0)) ; rm squish.log
for target in $*
do
   if ((jobcount+=1 > 18))
      then ((niceness = 18 ))
   else
      ((niceness = jobcount ))
   fi
   ((jobcount % 18 != 0)) || sleep 60
   nice -${niceness} compress ${target} && print "Finished compressing " \
      ${target}>> squish.log &
   print "Started compressing "${target} "at niceness " \
      ${niceness} >> squish.log
done
print "finished launching jobs" >> squish.log
```

A concurrently running *squeeze* process is started for each file. However, if run on a large directory, this could overload the system: therefore, *squeeze* uses **nice**(C) to decrease the priority of processes as the number increases.

The first section of this script keeps track of the niceness (decrement in scheduling priority) with which each *squeeze* job is to be started:

```
if ((jobcount+=1 > 18))
   then ((niceness = 18 ))
else
   ((niceness = jobcount ))
fi
```

The value of **jobcount** is incremented every time a new file compression job is started. If it exceeds 18, then the niceness value is pegged to 18; otherwise, the niceness is equal to the number of files processed so far. (**nice** accepts a maximum value of 18; this construct places a bounds check on the argument passed to it.)

The following line is a special test:

```
((jobcount % 18 != 0)) || sleep 60
```

If **jobcount** is *not* a multiple of 18 (that is, if there is a nonzero remainder when **jobcount** is divided by 18) then the first statement evaluates to **TRUE** and the second statement (separated by the logical OR) is not executed. Conversely, when **jobcount** is an exact multiple of 18, the first statement is evaluated to "0 != 0", which is false. When the first statement fails, the second statement (**sleep 60**) is executed. Thus, on reaching every eighteenth file, the script sleeps for one minute to allow the earlier compression processes to complete.

The real action of the script is as follows:

```
nice -${niceness} compress ${target} && print "Finished compressing " \
   ${target}>> squish.log &
print "Started compressing "${target} "at niceness " \
   ${niceness} >> squish.log
```

nice is used to start a **compress** process for each target file with the niceness level predetermined by the counter in the **if** loop at the top of the program. A logical AND connective is used to print a message to the file *squish.log* when the compression job terminates; the whole command line is executed as a background job. The shell then executes the next line, which prints a start message to the logfile, almost certainly executing it before the compression process has begun. (This illustrates the asynchronous execution of processes.)

It is well worth examining the logfile left after running *squeeze* on the contents of a directory. This illustrates how concurrent execution of processes can provide a significant performance improvement over sequential execution, despite the apparent complexity of ensuring that a rapid proliferation of tasks does not bring the system to its knees.

You can adapt *squeeze* to run just about any simple filter job in parallel; simply define a function to do the operation you want, then use it to replace **compress**.

Useful routines

The following routines are not entire scripts, but may be useful in context.

Locking files

It is sometimes necessary to use a shell script that controls access to a shared resource; for example, a file which should only be written by one person at a time. The following skeleton code shows an appropriate wrapper for such a script:

```
trap "exit 1" 1 2 3 15
#
# trap is vital, otherwise we may loop infinitely
#
LOCKFILE="/tmp/$$.LCK"
OMASK=$(umask)
umask 777
until > $LOCKFILE
do
   sleep 1
done 2> /dev/null
umask $OMASK
# now we can write critical data safely, unless root

  .
  .
  .

# finished critical section
rm -f $LOCKFILE
```

The user's old **umask** value is saved in **OMASK**, and their **umask** is reset to 777; this means that any files the user creates will have no read, write or execute permissions.

LOCKFILE is the name (determined elsewhere in the script) of a lock file. While a lock file exists, only the owner of the file should be allowed to operate on the shared data. This is ensured by the **until** loop:

```
until > ${LOCKFILE}
do
   sleep 1
done 2> /dev/null
```

The value of **until** only becomes TRUE when it can create a lockfile; this can only happen when no other users of the script have created a lock. (The lock has no write permission for anyone other than its creating process.) If this condition is true, the script creates the empty ${**LOCKFILE**} and continues; if false, it sleeps for a second and tries again. Having acquired the lockfile, the script resets **umask** to the user's original file creation permissions.

Having acquired a lock file, it is now certain that anyone else trying to run the script at the same time will get as far as the loop but no further; it is therefore safe to work on the shared resource, knowing that nobody else is simultaneously using it and might accidentally overwrite the user's changes. After using the shared resource, it is important to delete the lockfile; if the lock file is left behind, nobody will be able to access the shared resource.

This kind of access locking is typically used to control databases or critical applications where it is unsafe to risk a race condition (where two processes try to update a shared resource concurrently, overwriting each other's changes).

Context sensitive scripts

Some programs, for example **ls**, have many options. Rather than require users to always specify the commonest options, **ls** has a number of links (alternative names). When you run **ls** it examines the parameter **$0**, which contains the name under which it was invoked, and uses the appropriate options. For example, **l** is equivalent to **ls -l**; **lc** is equivalent to **ls -c**, and so on.

Your scripts can behave the same way. For example:

```
# should check number and type of args here
case `basename $0` in
add)    expr $1 + $2
        ;;
subtract) expr $1 - $2
        ;;
multiply) expr $1 \* $2
        ;;
divide) expr $1 / $2
        ;;
*)
        echo "Unknown operation: $0" >&2
        exit 1
        ;;
esac
exit
```

This short script has four names; it can be invoked as **add**, **subtract**, **multiply** and **divide**. It takes two arguments, and evaluates them according to the name under which it was invoked. **basename** is used to remove any preceding path (which might prevent the **case** statement from matching anything). For example:

```
$ add 5 4
9
$ subtract 4 5
-1
$
```

The variable **$0** contains the name under which the script was invoked. By using links to the script (rather than four separate script files) we conserve the number of files needed. In addition, if it is necessary to alter the behavior of all the programs, you can alter just the core file and the change will be recognized by all the links to it.

As an alternative, we could write an application that used several command line tools to update a database, all of which were links to a single tool that behaved differently depending on the context in which it was invoked.

Chapter 12

Regular expressions

The title of this section may be unfamiliar to you, but if you have used the SCO OpenServer system, you have almost certainly used regular expressions. Regular expressions are used to find files in a directory or text in a file. They may be made up of literal characters (like a search string in **vi**) or of a more complex pattern that can match several different possible combinations of characters. In essence, regular expressions describe the form rather than the content of a text string.

Rather than *being* the exact string of characters which are to be matched, a regular expression *describes* the character sequence. It is common for a regular expression to match more than one possible sequence of characters.

This chapter explains the following:

Literal characters in regular expressions

The simplest regular expression is a series of letters and numbers, possibly including white space (tabs or space characters), that have no special meaning. Such a regular expression consists of "literals"; that is, normal letters, which match only an identical letter in the data being searched. For example:

This is a regular expression

When an editor searches for a literal regular expression, it can only score a "hit" if it finds exactly that sequence of characters in the data it is searching. The example regular expression above will not, for example, match the following string:

This is a
regular expression

because there is a newline in the middle of it which was not specified in the regular expression.

Metacharacters in regular expressions

Any character that has a special meaning to the shell is a "metacharacter". For example, some punctuation marks, such as the period (.) and question mark (?), have a special meaning in some contexts that will cause the shell to try interpreting them rather than just reading them. One set of metacharacters is used to group commands. See, for example, "Entering commands on the same line" (page 120) and "Running commands in a pipeline" (page 120).

In addition, there are two families of regular expression metacharacters, the "Wildcard characters" (this page) used for matching filenames, and the more complex "Editor regular expressions" (page 317) metacharacters, which are used to match text strings within files. The asterisk is a wildcard character denoting any string consisting of zero or more characters.

As we saw in Chapter 11, "Automating frequent tasks" (page 245), these simple patterns are expanded by the shell, not by the program **ls** used in this example. All the shells recognize the same family of wildcard characters.

The second family of regular expressions is much more complex, and is used by such programs as **ed**(C), **sed**(C), **awk**(C), **vi**(C), **Tcl**(TCL), **grep**(C), and **egrep**(C). The editor regular expressions are used to search for text in files, rather than to search for files in directories. They are explained in "Editor regular expressions" (page 317).

Wildcard characters

Wildcards are used to match filenames. In addition to literal filenames, the shells recognize the following simple regular expressions:

*　　　　Matches any string of characters, including the null string (nothing). Thus, **foo*** matches "foot", "football" and "foo".

　　　　For example, **ls** * will match all the files in the current directory and its subdirectories. (The shell expands the " * " pattern; the **ls** command displays the results.)

ls g*∗ will match all files beginning with the letter " g ". In this case, the shell interprets the regular expression as meaning "any string of zero or more characters following a letter "g".

(Note that by convention, "dot" files such as *.profile* are excluded from such listings. In order to display these, it is necessary to use the **ls** command's **-a** option.)

?   Matches any single character. Thus, **foo?s** will match "foods" or "fools" but not "footballs". To match all files with a name comprising four characters type **ls ????**.

[...]   Matches one of the characters enclosed in brackets. (This is similar to " ? ", but restricted to the specified set of characters.) For example, the pattern **[Aa]** matches only the letters "A" and "a". **ls [Aa]ardvark** will match files called "aardvark" and "Aardvark". This is a useful construction in the light of the system's case-sensitivity.

A pair of characters separated by a " - " is taken to be a range. For example, **[A-Z]** is equivalent to **[ABCDEFGHIJKLM-NOPQRSTUVWXYZ]** and **ls [a-m]**∗ will match all the files beginning with the letters " a " to " m ". If the first character after the opening bracket is an exclamation mark (!), then any character *not* enclosed in the brackets is matched. For example, **[!0-9]** will match any character except a digit.

ls [!a-m]∗ will match all the files that do not begin with letters " a " through " m ".

Because the hyphen has a special meaning in a set, you can match a literal hyphen within a set only by placing it at the beginning or the end of the set. For example, **ls [abcde-]**∗ will match files beginning with **a**—**e** or **-**.

For more information about wildcard regular expressions, see **regexp**(M).

Editor regular expressions

Wildcard regular expressions are useful for selecting files, but they cannot search the text within files. For that, you need to use the editor regular expressions. These are as follows:

.   Matches any single character.

This is equivalent to the wildcard " ? ". For example, **.iddle** will match "diddle", "middle", or any other word beginning with some letter followed by the string "iddle".

∗   Matches zero or more repeating instances of the regular expression immediately preceding it. (See also " ? " below.)

For example, .*iddle* matches:

iddle
middle
twiddle

As a single character is taken to be a literal regular expression matching only itself, this means that a character followed by an asterisk matches zero or more instances of itself. Consequently, ".*" matches zero or more repeating instances of any character, and "a*" matches zero or more "a"s in a row.

Note that this behavior is not the same as that of the asterisk wildcard character. The shell interprets the asterisk wildcard to mean "zero or more *characters*"; in an editor regular expression, the asterisk matches zero or more instances of the *preceding regular expression*.

?   Matches zero or one occurrences of the regular expression immediately preceding it.

Note that, like the asterisk, this editor regular expression metacharacter does not have the same effect as its wildcard counterpart, which matches a single character, not an instance of a preceding regular expression.

+   Matches one or more (but not zero) occurrences of the regular expression immediately preceding it. (This feature is not available to all of the editor programs: see "Regular expression summary" (page 321).)

There is a subtle difference between the interpretation of regular expressions containing a " * " and a " + ". For example, suppose we have the word list:

fred
frog
figment
fuddled
ford

The expression "fr+" will match only "fred" and "frog", because it is constrained to match an "f" followed by at least one "r". However, "fr*" will match *all* of these words, because it matches an "f" followed by zero or more instances of the letter "r".

[...]   Matches any one of the characters enclosed in the brackets. If the first character in the set is a circumflex (^), it matches any one character that is *not* in the set. A hyphen between two characters in the set indicates a range; for example, [a-d] matches the first four letters of the alphabet. You can only include a literal closing bracket (]) in a class if it is the first character after the opening bracket.

If you are not certain of the spelling of a word that you are searching for, this construction comes in handy. For example, **rel[ae]v[ae]nt** matches any of:

relavant
relavent
relevant
relevent

^   Matches the beginning of a line if specified at the beginning of a regular expression; otherwise, it matches itself. The following specification uses ^ as a metacharacter:

^This is a nightmare

In the next specification, the ^ is a literal:

The ^ character is octal ASCII 136

$ Matches the end of a line if specified at the end of a regular expression; otherwise, it matches itself.

In the following, the dollar is used to match a string occurring at the end of a line:

It's the end of the line, folks$

In the next example, $ is a literal:

He stole $50000

\{*n,m*\}   Matches a range of occurrences of the regular expression immediately preceding it. *n* and *m* are positive decimal integers between 0 and 256. For example, \{5\} matches exactly five occurrences of the preceding expression, \{5,\} matches five or more occurrences of the preceding expression, and \{5,10\} matches between five and ten occurrences.

Escaping metacharacters

The special meaning of some metacharacters is dependent on their position in the regular expression, for example the start and end of line indicators. However, most metacharacters retain their special meaning irrespective of location. How then can they used as literal characters?

In such cases, the backslash (\) is used is used to "escape" the special meaning of the character that follows it. For example, \$ matches the "$" symbol rather than the end of a line. For example:

Summer sale now on\. Save $$\$

The backslash is used to quote the period and the final dollar sign (the other dollar signs are position-sensitive, and have no special meaning).

Because the backslash itself may be required to have only its literal meaning, \\ matches " \ ": the first backslash removes the special meaning from the second.

Regular expression grouping

Terms in an editor regular expression can be grouped together using \(and \). Any regular expression so constructed is treated as an identifiable unit in a larger regular expression, and can be referred to later in a search/replace expression by the editor.

This is a particularly useful mechanism. Each regular expression enclosed between escaped brackets is treated as a positional parameter. For example, in the regular expression \([Tt]he\).*\(fox\) the first grouped expression matches the words "The" or "the". It is followed by an indeterminate string of any characters, then a second grouped expression matching only the word "fox".

The first grouped expression may be referred to in the editor expression as \1, the second expression as \2, and so on. For an illustration of how this can be used to swap the order of regular expressions during a search and replace operation, see Chapter 14, "Manipulating text with sed" (page 371).

Grouping can be used to search for words separated by white space (tabs or spaces). For example, suppose you want to search for the expression above, where the words are separated by white space. You could construct a pattern like this:

  \([Tt]he\)\([⟨Tab⟩⟨Enter⟩⟨Space⟩]\{1,100\}\)\(fox\)

The middle group, \([⟨Tab⟩⟨Enter⟩⟨Space⟩]\{1,100\}\), is a group consisting of the set of space, tab and newline characters, matched from one to one hundred times. Thus, it will match from one to one hundred white space characters as a group separating "The" and "fox".

When a program that uses regular expressions tries to find a match, it searches for a string that matches the first group. If it finds a match, it then tries to match up the second group, then the third, and so on. A complete match is only confirmed when all the expressions in a group are correctly matched to a string of consecutive characters in the target file.

Precedence in regular expressions

Occasionally circumstances arise where a regular expression can match two or more strings in a target. In general, the leftmost, then the longest, string is selected; that is: if two matches overlap, the one starting to the left is selected, and if two matches starting at the same character position exist then the longest one is selected.

Precedence in the way that regular expressions are resolved can be forced by using the ( ) grouping operator. For example,

John(Dixon)?

matches the regular expression "John" followed by zero or one instances of the regular expression "Dixon". You can use brackets in conjunction with the vertical bar to group alternatives. For example, **factor(ies I y)** matches the words "factory" and "factories" slightly more economically than the equivalent regular expression **factories I factory**. In the event that no " I "s are present and there is only one "*", "+", or "?", the effect is that the longest possible match is chosen. So "ab*", presented with "xabbbby", will match "abbbb". Note that if "ab*" is tried against "xabyabbbz", it will match "ab" just after "x", due to the begins-earliest rule.

The decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.

Regular expression summary

Not all of the editor regular expression constructions are recognized by all of the editor programs. The following table categorizes the most common meta-characters and the programs that use them.

Note also that the table covers only the editor regular expression constructions. The wildcard metacharacters are not entirely compatible with the editor regular expression set described below. It is therefore important to be clear about which program will interpret a regular expression. Programs like **awk** and **grep** require you to enclose a regular expression on the command line, intended as an argument, in quotes. If you do not, the shell will try to interpret it as a wildcard regular expression, passing on any results to the program. This can have unexpected results.

Note that **awk** and **Tcl** in particular provide powerful programming constructs that can be used to manipulate text, but which fall outside the scope of regular expressions as such.

In the following table, a "y" indicates that the command supports the notation.

12

Regular expressions syntax recognized by programs

Command	Regular expression supported?											
	.	*	[...]	\(...\)	\{...\}	?	+	|	()	^	$	\
grep	y	y	y	y	y					y	y	y
egrep	y	y	y	y		y	y	y	y	y	y	y
awk	y	y	y			y	y	y	y	y	y	y
Tcl	y	y	y	y		y	y	y	y	y	y	y
ed	y	y	y	y	y					y	y	y
vi	y	y	y	y						y	y	y
sed	y	y	y	y	y					y	y	y

Note that due to subtle differences in the way the metacharacters are used by the editor programs, it is advisable to check the documentation that accompanies those programs. See, for example, Chapter 13, "Using awk" (page 323) and Chapter 14, "Manipulating text with sed" (page 371).

Korn shell regular expressions

In addition to the regular expression notations discussed above, the Korn Shell provides its own syntax. The metacharacters used by this syntax are similar to those used by the generic regular expression handling notations:

Operator	Meaning
*(*regexp*)	matches 0 or more instances of *regexp*
+(*regexp*)	matches 1 or more instances of *regexp*
?(*regexp*)	matches 0 or 1 instances of *regexp*
@(*regexp1* | *regexp2* |...)	matches *regexp1* or *regexp2* or ...
!(*regexp*)	matches any string except *regexp*

The OR notation given for the @ metacharacter can be combined with any of the other metacharacters. For example, **@(apple | pear | kumquat)** matches "apple" or "pear" or "kumquat", while **!(apple | pear | kumquat)** matches any string except "apple" or "pear" or "kumquat".

The **awk**(C) programming language is designed for processing and reporting on the contents of text files. Using **awk**, you can tabulate survey results, generate form letters, or reformat data files. The name **awk** is an acronym constructed from the initials of its developers (Aho, Weinberger, and Kernighan); it denotes the language and also the command you use to run an **awk** program.

awk does several useful things that you have to program for yourself in other languages. As a result, many **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping (that is, for writing quick prototypes of programs that will later be converted into a compiled language).

This chapter explains the following:

13

- using awk with other commands and the shell (page 362)
- spanning multiple lines (page 364)
- example applications (page 367)

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Fields

Normally, **awk** reads its input one line, or record, at a time. A record is, by default, a sequence of characters ending with a newline character. **awk** then splits each record into fields; by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the file *countries*. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is found. (Data is from 1978; the CIS (former USSR) has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank space separates both "North" and "South" from "America". The following example displays the contents of an input file:

```
CIS         8650    262     Asia
Canada      3852    24      North America
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
Australia   2968    14      Australia
India       1269    637     Asia
Argentina   1072    26      South America
Sudan        968    19      Africa
Algeria      920    18      Africa
```

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so the first record of *countries* has four fields, the second five, and so on. It is possible to set the field separator to just tab, so each line has four fields, matching the meaning of the data. We explain how to do this shortly. For the time being, let's use the default: fields separated by blanks or tabs. The first field within a line is called **$1**, the second **$2**, and so forth. The entire record is called **$0**.

Program structure

awk programs consist of a series of *patterns*, each of which is associated with an *action*. Each line of input is checked against each of the patterns in turn. For each pattern that matches, the associated action (which can involve multiple steps) is executed. Then the next line is read, and the matching starts over. This process typically continues until all the input has been read.

Patterns may be regular expressions or other, more complex entities.

So, the basic structure of an awk program is as follows:

```
pattern    { action }
pattern    { action }
. . .
```

For example:

```
$1 == "address"   { print $2, $3 }
```

This program prints the second and third fields of each input line whose first field is *address*.

Either the pattern or the action in a pattern-action statement can be omitted. If there is no action with a pattern, the matching line is printed. For example:

```
$1 == "name"
```

If there is no pattern with an action, the action is performed for every input line. For example:

```
{ print $1, $2 }
```

Because patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns. Use of fields and field notation is described in more detail in "Field variables" (page 327).

Running awk programs

There are two ways to run an awk program. First, you can type the command line to execute the pattern-action statements on the set of named input files:

awk *'pattern-action statements' optional_list_of_input_files*

For example, enter the following:

awk '{ print $1, $2 }' file1 file2

This program prints the first and second fields of every line in *file1* and *file2*.

When printed, items separated by a comma in the **print** statement are separated by the output field separator (by default, a single blank). Each line printed is terminated by the output record separator (by default, a newline).

If no fields are specified with a **print** command, **print** prints **$0**, the current record.

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk** reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (-) as one of the input files. For example, to read input first from *file1* and then from the standard input, enter the following:

> **awk '{ print $3, $4 }' file1 -**

This arrangement is convenient when the **awk** program is short. If the program is long, it is often more sensible to put it into a separate file and use the **-f** option to fetch it, as follows:

> **awk -f** *program_file optional_list_of_input_files*

You create an **awk** program the same way that you create a shell script; using a text editor such as **vi**. (Because **awk** programs are not directly executed, there is no need to set the executable permission bit on the file.) For example, the following command line specifies to fetch and execute the file *myprogram* on input from the file *file1*:

> **awk -f myprogram file1**

In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. In an example, if no input is mentioned, the input is assumed to be the file *countries*.

Formatting awk output

For more carefully formatted output, **awk** provides a C-like **printf** statement:

> **printf** *format, expr1, expr2, . . . , exprn*

This statement prints each *expr* according to the corresponding specification in the string *format*. For example, the following **awk** program:

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field ($1) as a string of 10 characters (right justified), then a space, then the third field ($3) as a decimal number in a six-character field, and finally a newline (\n). With input from the file *countries*, this program prints an aligned table:

```
CIS              262
Canada            24
China            866
USA              219
Brazil           116
Australia         14
India            637
Argentina         26
Sudan             19
Algeria           18
```

With **printf**, no output separators or newlines are produced automatically; you must create them yourself by using \n in the format specification. See "The printf statement" (page 355) for a full description of **printf**.

Variables

Unlike variables in C, **awk** variables do not need to be declared; that is, the type of information stored need not be defined beforehand. By default, **awk** variables have both a character string value and a numeric value: the appropriate one is derived from the context. Variable names must not contain spaces or periods.

The following sections describe the various variable types supported.

Field variables

Given that much of the work you will do with **awk** will involve the processing of records, **awk** provides a notation for the fast and efficient identification of fields. The fields of the current record are referred to by the *field variables* **$1, $2, ... , $NF**. Field variables share all of the properties of other variables: they can be used in arithmetic or string operations, and they can have values assigned to them. So, for example, you can divide the second field of the file *countries* by 1000 to convert the area from thousands to millions of square miles:

```
{ $2 /= 1000; print }
```

You can also assign a new string to a field:

```
$4 == "Africa"    { $4 = "South" }
```

Fields can be accessed by expressions. For example, **$(NF-1)** is the second to last field of the current record. For example:

```
$4 ~/Asia/ { print $(NF-1) }
```

This program prints the penultimate field (population) for each record in the file *countries* whose fourth field contains the string "Asia". (Omitting the parentheses causes a series of strings reading "-1" to be printed.)

A field variable referring to a nonexistent field, for example, **$(NF+1)**, has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file *countries* creates a fifth field giving the population density:

```
BEGIN  { FS = OFS = "\t" }
       { $5 = 1000 * $3 / $2; print }
```

This program adds a fifth column to the output. In the case of Canada, this would read "6.23053".

The number of fields may vary from record to record, but there is a limit of 100 fields per record.

Built-in variables

Besides reading the input and splitting it into fields, **awk** counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The built-in variable **NR** is the number of the current record, and **NF** is the number of fields in the record. For example, the following program prints the number of each line and how many fields it has:

```
{ print NR, NF }
```

This program prints each record preceded by its record number:

```
{ print NR, $0 }
```

In addition to **NR**, **awk** supplies the built-in variables listed in Table 13-1, "awk internal variables" (this page).

Table 13-1   awk internal variables

Variable	Meaning	Default
ARGC	number of command-line arguments	—
ARGV	array of command-line arguments	—
FILENAME	name of current input file	—
FNR	record number in current file	—
FS	input field separator	space or tab
NF	number of fields in current record	—
NR	number of records read so far	—
OFMT	output format for numbers	%.6g
OFS	output field separator	space
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by **match**	—
RLENGTH	length of string matched by **match**	—
SUBSEP	subscript separator	"\034"

User-defined variables

awk also allows you to define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file *countries*:

```
{ sum = sum + $3 }
{ print "Total population is", sum, "million"
  print "Average population of", NR, "countries is", sum/NR }
```

In general, **awk** initializes variables with the string value "" and the numeric value 0. Accordingly, *sum* is set to zero before it is used.

As you can see, to refer to a variable in **awk** you do not need to prefix it with "$" unless it is a field variable.

The first action accumulates the population from the third field; the second action prints the sum and average:

```
Total population is 262 million
Average population of 1 countries is 262
. . .
Total population is 2201 million
Average population of 10 countries is 220.1
```

Number or string?

Variables, fields, and expressions can have a numeric value, a string value, or both at any time. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like the following:

```
pop += $3
```

pop and **$3** must be treated numerically, so it must be ensured that their values are of the numeric type when arithmetic operations are performed on them. This is often called *type coercion*.

Similarly, in the following string, **$1** and **$2** must be strings, so they can be type coerced if concatenation operations are to be performed.

```
print $1 ":" $2
```

In an ambiguous context like the following, the type of the comparison depends on whether the fields are numeric or string:

```
$1 == $2
```

This can only be determined when the program runs; it might differ from record to record. For example, the above comparison succeeds on any pair of the following inputs:

> **1 1.0 +1 0.1e+1 10E-1 001**

However, it fails on the following inputs:

> **(null) 0**
> **(null) 0.0**
> **0a 0**
> **1e50 1.0e50**

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. The determination of type is done at run time.

There are two idioms for coercing an expression of one type to the other:

number ""   concatenate a null string to a *number* to coerce it to type string

string + 0   add zero to a *string* to coerce it to type numeric

Thus, the following forces a string comparison between two fields:

```
$1 " " == $2 " "
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**. **OFMT** specifies a **printf**-style format. By default, **OFMT** is **%.6g** (truncate argument to six digits). For example:

```
{pi=3.1415926535; print pi}
```

This prints the following:

```
3.14159
```

To print pi to 10 significant digits, use the format specifier **"%.10f"**, as follows:

```
BEGIN {OFMT="%.10f"}
        {pi=3.1415926535; print pi}
```

This program outputs the following:

```
3.1415926535
```

See **printf**(S) for details of the format string syntax.

A handful of useful one-liners

Although you can use **awk** to write large programs of some complexity, many programs are not much more complicated than what we have seen so far. Here is a collection of other short programs that you might find useful and instructive.

Print last field of each input line:

```
{ print $NF }
```

Print the tenth input line:

```
NR == 10
```

Print the last input line:

```
        { line = $0}
END     { print line }
```

Print input lines that do not have four fields:

```
NF != 4    { print $0, "does not have 4 fields" }
```

Print the input lines with more than four fields:

```
NF > 4
```

Print the input lines with last field more than 4:

```
$NF > 4
```

Print the total number of input lines:

```
END    { print NR }
```

Print total number of fields:

```
        { nf = nf + NF }
END     { print nf }
```

Print total number of input characters:

```
        { nc = nc + length($0) }
END     { print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/  { nlines++ }
END     { print nlines }
```

(The statement **nlines++** has the same effect as **nlines = nlines + 1**.)

Error messages

Generally, if you make an error in your **awk** program, you get an error message. For example:

```
$3 < 200 { print ( $1 }
```

This program generates the following error messages:

```
awk: syntax error at source line 1
  context is
          $3 < 200 { print ( $1 >>> } <<<
awk: illegal statement at source line 1
          1 extra (
```

Some errors might be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (**NR**) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that can be used as patterns.

Using simple patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called *relational operators* that make comparisons. As an example, the operator == tests for equality. Thus, to print the lines in which the fourth field equals the string "Asia", use the program consisting of the following single pattern:

```
$4 == "Asia"
```

With the file *countries* as input, this program yields:

```
CIS          8650     262      Asia
China        3692     866      Asia
India        1269     637      Asia
```

The complete set of comparisons are >, >=, <, <=, == (equal to), != (not equal to), ~ (matches), and ~! (does not match). These comparisons can be used to test both numbers and strings. For example, suppose you want to print only countries with a population greater than 100 million. All you need is the following program:

```
$3 > 100
```

(Remember that the third field in the file *countries* is the population in millions.) This program prints all lines in which the third field exceeds 100.

Second, you can use *regular expressions* that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes, as follows:

```
/US/
```

This program prints each line that contains the (adjacent) letters US anywhere; with the file *countries* as input, it prints the following:

```
USA     3615    219     North America
```

Third, you can use two special patterns, **BEGIN** and **END**, that match before the first record is read and after the last record is processed. The following program uses **BEGIN** to print a title:

```
BEGIN   { print "Countries of Asia:\n" }
/Asia/ { print "      ", $1 }
```

The output from this program is as follows:

```
Countries of Asia:
        CIS
        China
        India
```

BEGIN and END

BEGIN and **END** are two special patterns that give you a way of controlling initialization and wrap-up in an **awk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once, before the **awk** command starts to read its first input record. The **END** pattern matches the end of the input, after the last record has been processed.

The following **awk** program uses **BEGIN** to set the field separator to tab (**\t**) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can continue after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
               "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
        area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file *countries* as input, this program produces the following output:

```
COUNTRY          AREA        POP    CONTINENT
CIS              8650        262    Asia
Canada           3852        24     North America
China            3692        866    Asia
USA              3615        219    North America
Brazil           3286        116    South America
Australia        2968        14     Australia
India            1269        637    Asia
Argentina        1072        26     South America
Sudan             968        19     Africa
Algeria           920        18     Africa

TOTAL           30292       2201
```

Relational operators

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. To make comparisons, **awk** includes six relational operators and two regular expression matching operators, ˜ (tilde) and !˜, (discussed in "Regular expressions" (page 335)). Table 13-2, "awk comparison operators" (this page) shows the relational operators and their meanings.

Table 13-2 awk comparison operators

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can determine what is intended. See "Number or string?" (page 329) for more information about this.) Thus, the following pattern selects lines where the third field exceeds 100:

```
$3>100
```

This program selects lines that begin with the letters "S" through "Z" (that is, lines with an ASCII value greater than or equal to "S"):

```
$1 >= "S"
```

The output looks like this:

```
USA          3615      219      North America
Sudan         968       19      Africa
USA          3615      219      North America
Sudan         968       19      Africa
```

In the absence of any other information, **awk** treats fields as strings. Thus, the following program compares the first and fourth fields as strings of characters:

```
$1 == $4
```

Using the file *countries* as input, this program prints the single line for which this test succeeds:

```
Australia   2968    14    Australia
```

If both fields appear to be numbers, **awk** performs the comparisons numerically.

Regular expressions

awk provides regular expressions for pattern matching; the syntax of UNIX system expressions is described in Chapter 12, "Regular expressions" (page 315).

The simplest regular expression is a string of characters matching only itself: that is, the string is a *literal*. In **awk**, a regular expression is typically enclosed within slashes in order to label it as a regular expression as opposed to an **awk** command, as follows:

```
/Asia/
```

This program points to all input records that contain the substring "Asia"; if a record contains "Asia" as part of a larger string like "Asian" or "Pan-Asiatic", it is also printed.

awk provides the full range of UNIX system regular expression metacharacters; see Chapter 12, "Regular expressions" (page 315) for a detailed explanation. (In addition, **awk** recognizes the escape sequences listed in "The echo command" (page 254).) **awk** also provides the regular expression operators shown in Table 13-3, "awk regular expression operators" (this page).

Table 13-3 awk regular expression operators

Operator	Meaning
~	matches
!~	does not match

To restrict a match to a specific field, you use the matching operators ˜ (matches) and !˜ (does not match). The following program prints the first field of all lines in which the fourth field matches "Asia":

```
$4 ~ /Asia/ { print $1 }
```

This program prints the first field of all lines in which the fourth field does *not* match "Asia":

```
$4 !~ /Asia/ { print $1 }
```

awk interprets any string or variable on the right side of a ˜ or !˜ as a regular expression. For example:

```
$2 !~ /^[0-9]+$/
```

This sample program can be rewritten as follows:

```
BEGIN     { digits = "^[0-9]+$" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters such as ^[0-9]+$. When a literal quoted string like "^[0-9]+$" is used as a regular expression, one extra level of backslashes is needed to protect regular expression meta-characters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing "b" followed by a dollar sign. The regular expression for this pattern is **b\$**. To create a string to represent this regular expression, add one more backslash, as follows:

```
"b\\$"
```

The two regular expressions on each of the following lines are equivalent:

```
x ~ "b\\$"      x ~ /b\$/
x ~ "b\$"       x ~ /b$/
x ~ "b$"        x ~ /b$/
x ~ "\\t"       x ~ /\t/
```

A summary of the regular expressions and the substrings they match is given in Table 13-4, "awk regular expressions" (page 337). The unary operators *, +, and ? have the highest precedence, with concatenation next, and then alternation (|). All operators are left-associative. The *r* stands for any regular expression.

Table 13-4   awk regular expressions

Expression	Matches
char	any non-metacharacter *char*
char	character *char* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
r*	zero or more *r*s
r+	one or more *r*s
r?	zero or one *r*
(*r*)	*r*
r1 r2	*r1* then *r2* (concatenation)
r1 | *r2*	*r1* or *r2* (alternation)

Combining patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (OR), && (AND), and ! (NOT). For example, if you want to print all countries in Asia with a population of more than 500 million, use the following program:

```
$4 == "Asia" && $3 > 500
```

This program selects all lines in which the fourth field is **Asia** and the third field exceeds 500.

The following program selects lines with the strings "Asia" or "Africa" as the fourth field:

```
$4 == "Asia" || $4 == "Africa"
```

Another way to write the latter query is to use a regular expression with the alternation operator |:

```
$4 ~ /(Asia|Africa)/
```

The negation operator ! has the highest precedence, then &&, and finally ||. The operators && and || evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern ranges

A pattern range consists of two patterns separated by a comma:

> *pattern1, pattern2* { ... }

In this case, the action is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* (inclusive). As an example, the pattern **/Canada/, /Brazil/** matches lines starting with the first line that contains the string "Canada", up through the next occurrence of the string "Brazil":

```
Canada      3852      24     North America
China       3692      866    Asia
USA         3615      219    North America
Brazil      3286      116    South America
```

Similarly, because **FNR** is a built-in variable that represents the number of the current record in the current input file (and *FILENAME* is the name of the current input file), the following program prints the first five records of each input file with the name of the current input file prepended:

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they can also be a combination of one or more statements. This section describes the statements that can make up actions.

Performing arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file *countries*. Because the second field is the area in thousands of square miles, and the third field is the population in millions, the expression **1000 * $3 / $2** gives the population density in people per square mile. Use the following program to print the name of each country and its population density:

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

The output looks like this:

```
CIS             30.3
Canada           6.2
China          234.6
USA             60.6
Brazil          35.3
Australia        4.7
India          502.0
Argentina       24.3
Sudan           19.6
Algeria         19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +, -, *, /, % (remainder), and ^ (exponentiation; ** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **awk** recognizes and produces scientific (exponential) notation: 1e6, 1E6, 10e5, and 1000000 are numerically equal.

awk has assignment statements like those found in the C programming language. The simplest form is the assignment statement:

$v = e$

where v is a variable or field name, and e is an expression. For example, to compute the number of Asian countries and their total populations, use this program:

```
$4 == "Asia"  { pop = pop + $3; n = n + 1 }
END           { print "population of", n,
                      "Asian countries in millions is", pop }
```

Applied to *countries,* this program produces the following:

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern **$4 == "Asia"** contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **awk** initializes each variable with the string value " " and the numeric value 0.

The assignments in the previous program can be written more concisely using the operators **+=** and **++** as follows:

```
$4 == "Asia"    { pop += $3; ++n }
```

The += operator is borrowed from the C programming language:

```
pop += $3
```

It has the same effect as the following:

```
pop = pop + $3
```

The += operator is shorter and runs faster. The same is true of the ++ operator, which increments a variable by one.

The abbreviated assignment operators are +=, -=, *=, /=, %=, and ^=. These are shorthand versions of traditional operations: *a operator* = *b* has the same effect as *a* = *a operator b*.

The increment and decrement operators are ++ and --. As in C, you can use them as prefix (++x) or postfix (x++) operators. If x is 1, then i=++x increments x, then sets i to 2. On the other hand, i=x++ sets i to 1, then increments x. An analogous interpretation applies to prefix -- and postfix --. Assignment, increment, and decrement operators can all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3   { maxpop = $3; country = $1 }
END           { print country, maxpop }
```

Note that this program is not correct if all values of $3 are negative.

Functions

awk supplies a number of built-in arithmetic and string-handling functions.

Using arithmetic functions

awk provides the built-in arithmetic functions shown in Table 13-5, "awk arithmetic functions" (this page).

Table 13-5   awk arithmetic functions

Function	Value returned
atan2(y,x)	arctangent of $y/\hat{}x$ in the range $-\pi$ to π
cos(x)	cosine of x, with x in radians
exp(x)	exponential function of x
int(x)	integer part of x truncated towards 0
log(x)	natural logarithm of x
rand	random number between 0 and 1
sin(x)	sine of x, with x in radians
sqrt(x)	square root of x
srand(x)	x is new seed for **rand**

Both x and y are arbitrary expressions. The function **rand** returns a pseudo random floating point number in the range (0,1), and **srand** can be used to set the seed of the generator. If **srand** has no argument, the seed is derived from the time of day.

Using strings and string functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in "abc" or "hello, everyone". String constants can contain the C programming language escape sequences for special characters listed in "Regular expressions" (page 335).

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The following program prints each record preceded by its record number and a colon, with no blanks:

```
{ print NR ":" $0 }
```

This concatenates the three strings representing the record number, the colon, and the record, and prints the resulting string.

awk provides the built-in string functions shown in Table 13-6, "awk string functions" (this page). In this table, *r* represents a regular expression, *s* and *t* are string expressions, and *n* and *p* are integers.

Table 13-6   awk string functions

Function	Description
getline	reads next line of input
gsub(*r*, *s*)	substitutes *s* for *r* globally in current record, returns number of substitutions
gsub(*r*, *s*, *t*)	substitutes *s* for *r* globally in string *t*, returns number of substitutions
index(*s*, *t*)	returns position of string *t* in *s*, 0 if not present
length(*s*)	returns length of *s*
match(*s*, *r*)	returns the position in *s* where *r* occurs, 0 if not present; see built-in variables RSTART and RLENGTH
split(*s*, *a*)	splits *s* into array *a* on **FS**, returns number of fields
split(*s*, *a*, *r*)	splits *s* into array *a* on *r*, returns number of fields
sprintf(*fmt*, *expr-list*)	returns *expr-list* formatted according to format string *fmt*

(Continued on next page)

Table 13-6   awk string functions
(Continued)

Function	Description
sub(*r, s*)	substitutes *s* for first *r* in current record, returns number of substitutions
sub(*r, s, t*)	substitutes *s* for first *r* in *t*, returns number of substitutions
substr(*s, p*)	returns suffix of *s* starting at position *p*
substr(*s, p, n*)	returns substring of *s* of length *n* starting at position *p*
tolower(*s*)	returns *s* translated into lowercase
toupper(*s*)	returns *s* translated into uppercase

The **getline** function is used to read the next input line. Note that it does not return a value and that its syntax is like that of a statement: appending parentheses to it causes an error.

```
{ print "skipping record for ",$1
  getline
  print "going to record for ",$1 }
```

This code reads a record, prints the specified string, then executes the **getline** function which passes control onto the next record without processing:

```
skipping record for CIS
going to record for Canada
skipping record for China
. . .
```

For more information on **getline**, see "Multiline records and the getline function" (page 359).

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(C). The function **gsub**(*r, s, t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed**, the left-most match is used and is made as long as possible.) **gsub** returns the number of substitutions made. The function **gsub**(*r, s*) is a synonym for **gsub**(*r, s, $0*). For example, the following program transcribes its input, replacing occurrences of "USA" with "United States":

```
{ gsub(/USA/, "United States"); print }
```

Note that replacing the order of the commands in this action has an unexpected effect:

```
{ print gsub(/USA/, "United States",$0) }
```

The exit value of the operation as performed on each record is displayed:

```
0
0
0
0
1
0
0
```

In this case, only the fourth record of *countries* contains the string "USA": all other records return an exit value of 0.

The **sub** functions are similar to **gsub**, except that they only replace the first matching substring in the target string.

The function **index(s, t)** returns the left-most position where the string *t* begins in *s*, or zero if *t* does not occur in *s*. The first character in a string is at position 1. For example, the following command returns 2:

```
{ print index("banana", "an") }
```

The **length** function returns the number of characters in its argument string; thus, the following prints each record, preceded by its length:

```
{ print length($0), $0 }
```

(**$0** includes the input record separator but not the trailing newlines.) The following program prints the longest country name ("Australia"):

```
length($1) > max  { max = length($1); name = $1 }
END               { print name }
```

The **match(s, r)** function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function also sets two built-in variables **RSTART** and **RLENGTH**. **RSTART** is set to the starting position of the match in the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. (If a match does not occur, **RSTART** is 0, and **RLENGTH** is -1.) For example, the following program finds the first occurrence of the letter "i," followed by at most one character, followed by the letter "a" in a record:

```
{ if (match($0, /i.?a/))
      print RSTART, RLENGTH, $0 }
```

13

This program produces the following output from the file *countries*:

```
16     2     CIS         8650    262     Asia
26     3     Canada      3852    24      North America
3      3     China       3692    866     Asia
24     3     USA         3615    219     North America
27     3     Brazil      3286    116     South America
8      2     Australia   2968    14      Australia
4      2     India       1269    637     Asia
7      3     Argentina   1072    26      South America
17     3     Sudan       968     19      Africa
6      2     Algeria     920     18      Africa
```

Note that the **match** function matches the left-most longest matching string. For example, if you use the string "AsiaaaAsiaaaaan" as an input record, the following program matches the first string of **a**'s and sets **RSTART** to 4 and **RLENGTH** to 3:

```
{ if (match($0, /a+/))  print RSTART, RLENGTH, $0 }
```

Consider the following function:

sprintf(*format, expr1, expr2, …*)

returns (without printing) a string containing the following, formatted according to the **printf** specifications in the string *format*:

expr1, expr2, …, exprn

For a complete specification of these format conventions, see "The printf statement" (page 355).

The following statement assigns to x the string produced by formatting the values of **$1** and **$2**:

```
x = sprintf("%10s %6d", $1, $2)
```

It is assigned as a 10-character string and a decimal number in a field of width at least six; x can be used in any subsequent computation or display operation. For example:

```
{ x=sprintf("%10s%6d",$1,$2); print x }
```

This program produces the following output:

```
CIS             8650
Canada          3852
China           3692
USA             3615
Brazil          3286
Australia       2968
India           1269
Argentina       1072
Sudan            968
Algeria          920
CIS             8650
Canada          3852
China           3692
USA             3615
Brazil          3286
Australia       2968
India           1269
Argentina       1072
Sudan            968
Algeria          920
```

The function **substr**(s, p, n) returns the substring of s that begins at position p and is at most n characters long. If **substr**(s, p) is used, the substring goes to the end of s; that is, it consists of the suffix of s beginning at position p. For example, we could abbreviate the country names in *countries* to their first three characters by invoking the following program:

{ $1 = substr($1, 1, 3); print }

This produces the following output:

```
CIS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting **$1** in the program forces **awk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs. Attempting to change the setting of **OFS** back to a tab character with the command **{ OFS="\t" }** has the following result (only the first two lines are shown):

```
CIS     8650    262     Asia
Can     3852    24      North America
```

Note that this has had the undesirable effect of tab-separating "North" and "America" as well as the genuine fields.

Strings are stuck together (concatenated) by writing them one after another in an expression. For example, consider the following program:

```
      { s = s substr($1, 1, 3) " " }
END   { print s }
```

When invoked on the file *countries*, the program prints the following by building **s** up, one piece at a time, from an initially empty string:

```
CISCanChiUSABraAusIndArgSudAlg
```

Control flow statements

awk provides **if**, **if-else**, **while**, **do-while**, **for**, and **?** statements, and statement grouping with braces. The syntax of these constructs is similar to that of the C programming language, but their usage is similar to the shell constructs covered in Chapter 11, "Automating frequent tasks" (page 245). In particular, **if**, **if-else** and **?** are branching constructs and **while**, **do-while**, and **for** are looping constructs. There is no equivalent of the shell **case** and **select** statements.

if statements

The **if** statement's generic syntax is as follows:

> **if** (*expression*) *statement1*[**else** *statement2*]

The *expression* acting as the conditional has no restrictions; it can include the relational operators **<, <=, >, >=, ==,** and **!=;** the regular expression matching operators **~** and **!~** ; the logical operators **| |, &&,** and **!;** juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the *expression* is first evaluated. If it is nonzero and non-null, *statement1* is executed; otherwise *statement2* is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

The following is a rewrite of the maximum population program from "Performing arithmetic" (page 338), using an **if** statement:

```
BEGIN { minpop=1000; maxpop=0; mincountry=""; maxcountry="" }
        {     if (maxpop < $3) {
              maxpop = $3
              maxcountry = $1
        } else
        {     if (minpop > $3) {
                  minpop=$3
                  mincountry=$1
            }
        }
}
END   { print maxcountry, maxpop
        print mincountry, minpop }
```

The following output results:

```
China 866
Australia 14
```

while statements

The **while** statement is exactly that of the C programming language:

while (*expression*) *statement*

The *expression* is evaluated; if it is nonzero and non-null, the *statement* is executed, and the *expression* is tested again. The cycle repeats as long as the *expression* is nonzero. For example, use the following to print all input fields one per line:

```
{  i = 1
   while (i <= NF) {
       print $i
       i++
   }
}
```

The **do-while** statement has the following form:

do *statement* **while** (*expression*)

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for**, which test for completion at the top of the loop.

13

The following example of a **do-while** statement prints all lines except those occurring between the strings "start" and "stop":

```
/start/ {
            do {
        getline x
            } while (x !~ /stop/)
        }
        { print }
```

for statements

The **for** statement is like that of the C programming language rather than that of the shell:

for (*expression1* ; *expression2* ; *expression3*) *statement*

This has the same effect as the following:

expression1
while (*expression2*) {
    statement
    expression3
}

Thus, the following statement does the same job as the **while** example above:

{ **for** (i = 1; i <= NF; i++) **print** $i }

Flow control statements

The **break** statement causes an immediate exit from an enclosing **while** or **for**, thereby preventing further iterations from being performed. The **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action, if any, is executed. Within the **END** action, the following statement causes the program to return the value of *expr* as its exit status:

exit *expr*

If there is no *expr*, the exit status is zero.

Arrays

awk provides one-dimensional arrays. An array is a list of variables that share a common name, and that are distinguished from one another by a subscript (that is, a number indicating their position in the list). You do not need to declare arrays and array elements; like variables, they spring into existence when you use them. An array subscript can be a number or a string.

As an example of a conventional numeric subscript, the following statement assigns the current input line to the NRth element of the array x:

```
x[NR] = $0
```

In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the awk program like this:

```
        { x[NR] = $0 }
END    { ... processing ... }
```

The first action records each input line in the array *x*, indexed by line number; processing is done in the END statement.

Array elements can also be named by nonnumeric values. An array like this, where the array subscript (that is, position within the array of a given member) is a string, is called an *associative* array. For example, the following program accumulates the total population of Asia and Africa into the associative array pop. The END action prints the total population of these two continents.

```
/Asia/   { pop["Asia"] += $3 }
/Africa/        { pop["Africa"] += $3 }
END     { print "Asian population in millions is", pop["Asia"]
            print "African population in millions is",
         pop["Africa"] }
```

On the file *countries*, this program generates the following output:

```
Asian population in millions is 1765
African population in millions is 37
```

In this program, if we use pop[Asia] instead of pop["Asia"], the expression uses the value of the variable Asia as the subscript. Because the variable is uninitialized (does not exist), the values would have been accumulated in pop[""].

Suppose our task is to determine the total area in each continent of the file *countries*. Any expression can be used as a subscript in an array reference. Consider the following statement:

```
area[$4] += $2
```

This program uses the string in the fourth field of the current input record to index the array *area* and in that entry accumulates the value of the second field:

```
BEGIN           { FS = "\t" }
                { area[$4] += $2 }
END             { for (name in area)
                  print name, area[name] }
```

When you invoke this on the *countries* file, this program produces the following output:

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

Stipulating the **FS** character is necessary in order to prevent **awk** from inter-preting strings containing white space ("South America", for example) as two separate fields, in which case the following occurs:

```
Africa 1888
South 4358
Asia 13611
North 7467
Australia 2968
```

(Note that the order of the output fields is different from that of the previous example. This illustrates an important quality of associative arrays, namely that the elements in the array are not stored in any particular order, as is the case with conventional arrays. While numeric indices can be used in associa-tive arrays, they do not necessarily refer to sequentially ordered locations. In order to manipulate the elements sequentially, a loop must be established that will increment a pointer to the elements.)

The last example uses a form of the **for** statement that iterates over all defined subscripts of an array:

> **for** (*i* **in** *array*) *statement*

This executes *statement* with the variable *i* set in turn to each value of *i* for which **array[i]** has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when *i* or *array* is altered during the loop.

awk does not provide multidimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable **SUBSEP**). For example, the following code creates an array that behaves like a two-dimensional array; the subscript is the concatenation of **i**, **SUBSEP**, and **j**:

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
                arr[i,j] = . . .
```

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in **arr**:

```
if ("Africa" in area) . . .
```

This condition performs the test without the side effect of creating **area["Africa"]**, which would happen if we used the following:

```
if (area["Africa"] != "") . . .
```

Note that neither is a test of whether the array *area* contains an element with value **"Africa"**.

It is also possible to split any string into fields in the elements of an array using the built-in function **split** (see "Using strings and string functions" (page 341)). The function splits the string **s1:s2:s3** into three fields (using the separator :) and stores **s1** in **a[1]**, **s2** in **a[2]**, and **s3** in **a[3]**.

```
split("s1:s2:s3", a, ":")
```

The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element can be deleted with the **delete** statement:

> **delete** *arrayname*[*subscript*]

User-defined functions

awk provides user-defined functions, which are useful tools for extending the syntax of the language. For example, the following program defines and tests the usual recursive factorial function (using some input other than the file *countries*):

```
function fact(n) {
    if (n <= 1)
      return 1
    else
      return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

The function is defined at the start of the program, before it is used.

In the definition, **n** is a *formal* parameter: that is, it is used in the definition of the function and within the body of the function. When **fact( )** is used, the formal parameter **n** is replaced with an *actual* parameter. For example:

```
fact(mynum)
```

This statement prints the factorial of the actual parameter **mynum**. Thus, the formal parameter list is effectively a template into which you can slot your own actual parameters.

The command **return** returns the given value, so that the assignment:

```
result = fact(mynum)
```

causes **fact( )** to return the value to **result**. If no **return** command is used, **fact( )** is effectively valueless, so the assignment above would be meaningless.

A function is defined as:

> **function** *name* (*argument-list*){
> > *statements*
> **}**

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; these are called the *formal* parameters of the function. Within the body of the function, these variables refer to the *actual* parameters by which they are replaced when the function is called.

There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation.

Sometimes you may need to pass a large amount of data to a function; for example, an entire line of text. This is best accomplished by using an array as an argument, rather than by passing a set of individual variables. Individual variables, or scalars, are passed by value; that is, rather than the function having access to the variable itself, the function receives a copy of the argument. In contrast, array arguments are passed by reference: that is, the function can access the elements of the array directly (rather than a copy of the array which is local to the function, being deleted after the function terminates). Consequently it is possible for the function to alter array elements or create new ones that are accessible outside the function.

The difference is subtle. When a variable is passed by value, an internal copy of it is used within the function, so the function cannot affect the value of the argument outside its own scope. Consequently, if you have a variable **myvar** that is a parameter to a function, any changes you make to **myvar** within the function will be lost when the function returns.

In contrast, a variable that is passed by reference (like an array) is totally accessible both within the function and throughout the rest of the program.

Functions can access variables that are not passed as parameters. In general, variables created in an **awk** program are global (that is, accessible anywhere) unless they are the formal parameters to a function. Formal parameters are local, cannot be accessed outside the function, and are lost when the function exits. They also override existing variables of the same name when the function is being executed; if you declare a function with a formal parameter of the same name as an existing variable, references to the variable name within the function will only refer to the formal parameter. Any changes you make to their values are lost as soon as you exit the function.

You can have any number of extra formal parameters that are used purely as local variables; this is particularly useful if you want to perform some sort of internal process that you do not want to refer to anywhere else in the program.

Some lexical conventions

Comments may be placed in **awk** programs. They begin with the character "#" and end at the end of the line. For example:

```
print x, y      # this is a comment
```

Statements in an **awk** program normally occupy a single line. Several statements can occur on a single line if they are separated by semicolons. You can continue a long statement over several lines by terminating each continued line by a backslash. (It is not possible to continue a quoted string; you must close the quotes, add the backslash, then reopen the quotes on the next line.) This explicit continuation is rarely necessary, however, because statements continue automatically after the operators && and || or if the line ends with a comma (for example, as might occur in a **print** or **printf** statement).

Several pattern-action statements can appear on a single line if separated by semicolons.

awk output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output so that output from **print** and **printf** can be directed to files and pipes. This section describes how to use these two statements.

The print statement

The following statement prints the string value of each expression separated by the output field separator (**OFS**) followed by the output record separator (**ORS**):

> print *expr1, expr2, . . . , exprn*

The statement **print** is an abbreviation for the following:

```
print $0
```

To print an empty line, use the following:

```
print " "
```

Output separators

The built-in variables **OFS** and **ORS** contain the output field separator and record separator respectively. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but you can change these values at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN  { OFS = ":"; ORS = "\n\n" }
       { print $1, $2 }
```

Run on the *countries* file, the following output is generated (only the first few lines are shown):

```
CIS:8650


Canada:3852


China:3692
```

Notice that the following program prints the first and second fields with no intervening output field separator:

```
{ print $1 $2 }
```

This is because **$1 $2** is a string consisting of the concatenation of the first two fields. The output is as follows:

```
CIS8650


Canada3852
```

You may want to change the value of **OFS** or **ORS** if you are reading or writing a file used by an application that does not follow the SCO OpenServer convention of separating fields by tabs and records by newlines.For example, in data files created by the language BASIC, fields are surrounded by double quotes and separated by commas, while records are separated by newlines. Therefore, to create a data file in **awk** that could be used by a BASIC program, you would need to set **OFS** to be a comma, and remember to print all output fields surrounded by double quotes.

The printf statement

awk's **printf** statement is the same as that in C, except that the ∗ format specifier is not supported. The **printf** statement has the general form:

> printf *format, expr1, expr2, . . . , exprn*

where *format* is a string that contains both information to be printed and specifications on what conversions to perform on the expressions in the argument list, as in the table below. Each specification begins with a "%", ends with a letter that determines the conversion, and can include any of the following:

-        left-justify expression in its field

width   pad field to this width as needed; fields that begin with a leading 0 are padded with zeros

prec   maximum string width or digits to right of decimal point

Table 13-7   awk printf conversion characters

Character	Prints expression as
%c	single character
%d	decimal integer
%e	[-]d.d*precision*E[+-]dd
%f	**[-]ddd.d***precision*
%g	**e** or **f** conversion, whichever is shorter, with nonsignificant zeros suppressed
%o	unsigned octal number
%s	string
%x	unsigned hexadecimal number
%%	print a %; no argument is converted

13

Here are some examples of **printf** statements with the corresponding output:

```
printf "%d", 99/2          49
printf "%e", 99/2          4.950000e+01
printf "%f", 99/2          49.500000
printf "%6.2f", 99/2       49.50
printf "%g", 99/2          49.5
printf "%o", 99/2          61
printf "%06o", 99/2        000061
printf "%x", 99/2          31
printf "|%s|", "January"   |January|
printf "|%10s|", "January" |   January|
printf "|%-10s|", "January" |January   |
printf "|%.3s|", "January"  |Jan|
printf "|%10.3s|", "January" |       Jan|
printf "|%-10.3s|", "January" |Jan       |
printf "%%"                %
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into files

You can print output into files, instead of to the standard output, using the "**>**" and "**>>**" redirection operators. For example, if you invoke the following program on the file *countries*, **awk** prints all lines where the population (third field) is bigger than 100 into a file called *bigpop*, and all other lines into *smallpop*:

```
$3 > 100    { print $1, $3 >"bigpop" }
$3 <= 100   { print $1, $3 >"smallpop" }
```

Notice that the filenames must be quoted; without quotes, *bigpop* and *smallpop* are uninitialized variables. If the output filenames are created by an expression, they also must be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the **>** operator has higher precedence than concatenation; without parentheses, the concatenation of *tmp* and *FILENAME* does not work.

Note that files are opened once in an **awk** program. If **>** is used to open a file, its original contents are overwritten. But if **>>** is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output into pipes

You can also direct printing into a pipe with a command on the other end, instead of into a file. The following statement causes the output of **print** to be piped into the *command-line*:

print | "*command-line*"

Although we show the *command-line* and filenames here as literal strings enclosed in quotes, they can also come from variables, and the return values from functions.

Suppose we want to create a list of continent-population pairs, and sort it alphabetically by continent. The **awk** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called **pop**. Then it prints each continent and its population, and pipes this output into the **sort** command:

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (c in pop)
              print c ":" pop[c] | "sort" }
```

Invoked on the file *countries*, this program yields the following:

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named **sort**), but they are created and opened only once in the entire run. So, in the last example, for all **c** in **pop**, only one sort pipe is open.

In **awk**, only one pipe may be open at a time. In order to open a second pipe, the first must be closed. The statement **close**(*pipe*) closes a pipe, where *pipe* is the string used to create it in the first place. For example:

```
close("sort")
```

When opening or closing a file, different strings are different commands.

The special files */dev/stdout* and */dev/stderr* are particularly useful for printing error messages. To print to the standard error without using */dev/stderr*, it is necessary to create a pipe and catenate its output with the standard error of the current shell:

```
{ ... print "error message" | "cat >&2"; ... }
```

However, using the special files, it is possible to direct standard error and standard output messages appropriately:

```
{ ... print "error message" > "/dev/stderr"; ... }
```

Input

The most common way to give input to an **awk** program is to name on the command line the file that contains the input. This is the method that we have been using in this chapter. However, you can use several other methods, each of which is described in this section.

Files and pipes

You can provide input to an **awk** program by putting the input data into a file, say *awkdata*, and then executing it like this:

awk *'program' awkdata*

awk reads its standard input if no filenames are given; thus, a second common arrangement is to have another program pipe its output into **awk**. For example, **grep**(C) selects input lines containing a specified regular expression, but it can do so faster than **awk**, because this is the only thing it does.

```
grep 'Asia' countries | awk '{ print $1 }'
```

In this shell script, **grep** quickly finds the lines containing Asia and passes them on to the **awk** program for subsequent processing, where the first field is printed:

```
CIS
China
India
```

Input separators

With the default setting of the field separator **FS**, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1    field2
  field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. For example, the following sets it to any or all of comma, space and tab:

```
BEGIN { FS = "([, \\t])" }
```

You can also set **FS** on the command line with the **-F** argument. For example, this behaves the same as the previous example:

```
awk -F'([, \\t])'
```

Regular expressions used as field separators match the leftmost longest occurrences (as in the **sub** function), but they do not match null strings.

Multiline records

Records are normally separated by newlines, so that each line is a record; you can change this, but only in a limited way. You can set the built-in record separator variable **RS** to an empty string as follows:

```
BEGIN   { RS = " " }
```

Input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to set the record separator to an empty line and the field separator to a newline, as in the following example:

```
BEGIN   { RS = " "; FS = "\n" }
```

A record is limited to 3000 characters. See "Multiline records and the getline function" (this page) and "Cooperation with the shell" (page 362) for some other examples of processing multiline records.

Multiline records and the getline function

awk's facility for automatically breaking its input into records that are more than one line long is not adequate for all tasks. For example, if records are not separated by blank lines, but by something more complicated, setting **RS** to null does not work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

Use the function **getline** to read input either from the current input or from a file or pipe, by using redirection in a manner analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF**, **NR**, and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multiline records, each of which begins with a line beginning with **START** and ends with a line beginning with **STOP**. The following **awk** program processes these multiline records, a line at a time, putting the lines of the record into consecutive entries of an array:

f[1] f[2] ... f[nf]

Once the line containing **STOP** is encountered, the record can be processed from the data in the **f** array:

```
/^START/ {
          f[nf=1] = $0
          while (getline && $0 !~ /^STOP/)
               f[++nf] = $0
          # now process the data in f[1] ... f[nf]
          . . .
}
```

Notice that this code uses the fact that **&&** evaluates its operands left to right and stops as soon as one is true. The same job can also be done by the following program:

```
/^START/ && nf==0        { f[nf=1] = $0 }
nf > 1  { f[++nf] = $0 }
/^STOP/ { # now process the data in f[1] . . . f[nf]
          . . .
          nf = 0
}
```

The following statement reads the next record into the variable **x**:

```
getline x
```

No splitting is done; **NF** is not set.

This statement reads from *file* instead of the current input:

```
getline <"file"
```

It has no effect on **NR** or **FNR**, but field splitting is performed, and **NF** is set.

The following statement gets the next record from *file* into **x**:

```
getline x <"file"
```

In this case, no splitting is done, and **NF, NR,** and **FNR** are untouched.

Note that if a filename is an expression, it should be in parentheses for evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) {  ... }
```

(See "Command-line arguments" (page 361) for a discussion of **ARGV**.) This is because the < has precedence over concatenation. Without parentheses, a statement such as the following attempts to set *x* to read a file called *tmp$FILENAME*:

```
getline x < "tmp" FILENAME
```

Also, if you use this **getline** statement form, a statement like the following loops forever if the file cannot be read:

```
while ( getline x < file ) { ... }
```

This is because **getline** returns -1, not zero, if an error occurs. A better way to write this test is as follows:

```
while ( getline x < file > 0) { ... }
```

It is also possible to pipe the output of another command directly into **getline**. For example, the following statement executes **who** and pipes its output into **getline**:

```
while ("who" | getline)
        n++
```

Each iteration of the **while** loop reads one more line and increments the variable **n**, so after the **while** loop terminates, **n** contains a count of the number of users. Similarly, the following statement pipes the output of **date** into the variable **d**, thus setting **d** to the current date:

```
"date" | getline d
```

The table below summarizes the **getline** function.

Table 13-8   getline function

Form	Sets
getline	$0, NF, NR, FNR
getline *var*	*var*, NR, FNR
getline *<file*	$0, NF
getline *var <file*	*var*
cmd	getline	$0, NF
cmd	getline *var*	*var*

Command-line arguments

The command-line arguments are available to an **awk** program: the array **ARGV** contains the elements **ARGV[0]**, ..., **ARGV[ARGC-1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **awk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments).

The following command line (typed at the shell prompt) contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
    for (i = 1; i < ARGC; i++)
            printf "%s ", ARGV[i]
    printf "\n"
}' $*
```

You can modify or add to the arguments; you can also alter **ARGC**. As each input file ends, **awk** treats the next non-null element of **ARGV** (up to the current value of **ARGC-1**) as the name of the next input file.

The one exception to the rule that an argument is a filename is that if it is of the following form, then the variable *var* is set to the value *value* as if by assignment:

> *var=value*

If *value* is a string, no quotes are needed. Such an argument is not treated as a filename.

Using awk with other commands and the shell

awk gains its greatest power when you use it in conjunction with other programs. Here we describe some of the ways in which **awk** programs cooperate with other commands.

The system function

The built-in function **system**(*command-line*) executes the command *command-line*, which can be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed. The output from *command-line* is not automatically available for use within the **awk** script. It must be piped into **getline**, as follows:

> **system**(*command-line* | *getline*)

The following program calls the shell command **cat**(C) to print the file named in the second field of every input record whose first field is "#include," having first stripped any <, >, or " characters that might be present:

```
$1 == "#include" { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

Cooperation with the shell

In all the examples thus far, the **awk** program is in a file and is retrieved using the **-f** flag, or it appears on the command line enclosed in single quotes, as in the following example:

```
awk '{ print $1 }' ...
```

Since **awk** uses many of the same characters as the shell does, such as $ and ", surrounding the **awk** program with single quotes ensures that the shell passes the entire program unchanged to the **awk** interpreter.

Now, consider writing a command **addr** that searches a file *addresslist* for name, address, and telephone information. Suppose that *addresslist* contains names and addresses in which a typical entry is a multiline record such as the following:

> G. R. Emlin
> 600 Mountain Avenue
> Murray Hill, NJ 07974
> 201-555-1234

Records are separated by a single blank line.

You want to be able to search the address list by issuing commands like the following:

> **addr Emlin**

To do this, create a program of the following form:

```
awk '
BEGIN   { RS = " " }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called *addr* that contains the following lines:

```
awk '
BEGIN   { RS = " " }
/'$1'/
' addresslist
```

The quotes are critical here. The **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The **$1** is outside the quotes, visible to the shell, which then replaces it by the pattern *Emlin* when you invoke the command **addr Emlin**.

A second way to implement **addr** relies on the fact that the shell substitutes for $ parameters within double quotes:

```
awk "
BEGIN   { RS = \"\" }
/$1/
" addresslist
```

Here you must protect the quotes defining **RS** with backslashes so that the shell passes them on to **awk**, uninterpreted by the shell. **$1** is recognized as a parameter, however, so the shell replaces it by the pattern when you invoke the following command:

> **addr** *pattern*

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline**:

```
awk '
BEGIN    { RS = " "
             while (getline < "addresslist")
                if ($0 ~ ARGV[1])
                    print $0
} ' $*
```

All processing is done in the **BEGIN** action.

Notice that you can pass any regular expression to **addr**; in particular, you can retrieve parts of an address or telephone number, as well as a name.

Spanning multiple lines

Multiline records are a suitable solution for handling data that is regular in form, but sometimes it is necessary to handle records that have no fixed length. Under these circumstances, when the input data is irregular, setting **RS** and **FS** is not going to be very helpful.

An illustrative example of this kind of problem occurs in the example "Writing a readability analysis program: an example" (page 266); one of the tasks is to count the number of sentences in a text file. Although the file consists of lines of text terminated by a carriage return, there is no guarantee that a sentence occupies a single line. Sentences are started by a word with an initial capital letter and are terminated by a period: they may occupy one or more lines.

This kind of problem can be handled by using a variable as a buffer. A line is read in and appended to the buffer; the entire buffer is then searched to see if it contains a sentence. If no sentence is contained, another line is read, and so on. If a sentence is matched, it is counted and all the data in the buffer up to the end of the sentence is deleted.

For example:

```
#!/usr/bin/awk -f
BEGIN {
 init="(^[A-Za-z1-90][.])|([[:space:]]|[.])(([A-Za-z0-9]|[A-Za-z0-9][a-z0-9])[.])"
 sent="([A-Za-z1-90]+([[:space:]])*)+[.]"
 sentences = 0
 target = ""
 marker="+X+"
 }
 {   initials = gsub(init, "", $0)
 target = target " " $0
 hit = gsub(sent, marker, target)
 sentences += hit
 if (hit != 0) {
     for (i=0; i< hit; i++) {
         found = index(target, marker)
         target = substr(target, (found+3))
     } # end for
 } # end if
 hit = 0
 }
END {    print sentences " sentences counted"
  }
```

The **BEGIN** section is used to define a sentence (in variable **sent**). For the purpose of this program, a sentence is a regular expression consisting of a sequence of words terminated by a period. (A word is one or more letters followed by an optional space. Because **awk** matches the left-most longest pattern, in practice we can expect **awk** to choose the longest series of letters it can find that is terminated by a space.)

Sentences are not the only entities terminated by a period; initials and elipses contain periods, and must therefore be removed before the input is tested to determine if it is a sentence. The **BEGIN** section defines the variable **init** as a regular expression that matches a set of initials.

Initials consist of a letter or digit followed by a period, or a more complex format (one or two letters or digits followed by a period, as in Ph.D.). This expression is not infallible, but traps most initials.

Each line in the standard input (**$0**) is read in and scanned for initials. These are replaced by null characters (""). The line is then appended to the variable **target** (line 10 of the program).

13

On line 11, **target** is scanned and every occurrence of a sentence (as defined by **sent**) is replaced by a marker (defined by **marker**). The total sentence count is incremented by the number of sentences found in **target**. Then, for each hit, the **target** variable is truncated; the text prior to the last marker is discarded. (That is, the matched sentences are removed from **target**, by the **substr** command on line 16.) The program then reads the next line. At the end of the input, the script displays a total sentence count.

It is worth comparing this method of crossing line boundaries with the method using **sed** in "Hold and get functions" (page 385).

Syllables are handled in a similar manner, but it is not necessary to handle multiple lines. Instead, a regular expression that matches a generic syllable is defined, and a simple loop globally replaces all syllables with a marker character while incrementing a counter:

```
#!/bin/sh
#
# First, define syllabic consonants
#
CONS="[bcdfghjklmnpqrstvwxyz]|ll|ght|qu|([wstgpc]h)|sch"
#
# Next, define syllabic vowels
#
VOWL="[aeiou]+|ly"
#
# The definition of a syllable (after Webster's Collegiate Dictionary):
# a syllable is one or more consonants or vowels, optionally preceded by
# and optionally followed by a consonant.
#
SYL="(${CONS})*\
((${CONS})|((${VOWL})+))\
(${CONS})*"
#
sylcount=`awk -e ' BEGIN { sylcount = 0 }
                         { target = $0
                           incr = gsub(syllable, "*", target)
                           sylcount += incr
                         }
                  END    { print sylcount }
' syllable="$SYL" < $1`
echo "There were $sylcount syllables in $1"
```

Note that for the purposes of matching a syllable, we need to use syllabic consonants and syllabic vowels. These correspond to the written representations of parts of speech, rather than to the letters of the alphabet. Therefore, the syllable definition given above is so complex that it is better to build it up from component expressions stored in environment variables (as above) than to try to write it out at length.

After processing the specified input file ($1), the script displays a count of all the syllables located.

Example applications

awk has been used in surprising ways. We have seen **awk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages, such as Pascal or C. In this section, we present a few more examples to illustrate some additional **awk** programs.

Generating reports

awk is especially useful for producing reports that summarize and format information. Suppose you want to produce a report from the file *countries*, that lists the continents alphabetically, and after each continent, its countries in decreasing order of population, like this:

```
Africa:
        Sudan          19
        Algeria        18

Asia:
        China          866
        India          637
        CIS            262

Australia:
        Australia       14

North America:
        USA            219
        Canada          24

South America:
        Brazil         116
        Argentina       26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, create a list of continent-country-population triples, in which each field is separated by a colon. To do this, use the following program, **triples**, which uses an array **pop**, indexed by subscripts of the form 'continent:country' to store the population of a given country.

The **print** statement in the **END** section of the program creates the list of continent-country-population triples that are piped to the **sort** routine:

```
BEGIN   { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END     { for (cc in pop)
               print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for **sort** deserve special mention. The **-t:** argument tells **sort** to use : as its field separator. The **+0 -1** arguments make the first field the primary sort key. In general, **+i -j** makes fields **i+1**, **i+2**, ..., **j** the sort key. If **-j** is omitted, the fields from **i+1** to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (**n** is for numeric, **r** for reverse order). Invoked on the file *countries*, this program produces as output:

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:CIS:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form, run it through a second **awk** program, **format**:

```
BEGIN   { FS = ":" }
{       if ($1 != prev) {
             print "\n" $1 ":"
             prev = $1
        }
        printf "\t\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The following command line produces the report:

awk -f triples countries | awk -f format

As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awk** and **sort** operations.

Word frequencies

Here we show how to use associative arrays for counting. Suppose you want to count the number of times each word appears in the input, where a word equals any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order:

```
        { for (w = 1; w <= NF; w++) count[$w]++ }
END  { for (w in count) print count[w], w | "sort -nr" }
```

The first statement uses the array **count** to accumulate the number of times each word is used. Once the input has been read, the second **for** loop pipes the final count, along with each word, into the **sort** command. Running this program on the first two paragraphs of this chapter produces output that starts as follows:

```
6 awk
4 the
4 programs
4 for
4 and
3 you
. . .
```

Accumulation

Suppose you have two files of records, *deposits* and *withdrawals*, that contain a name field and an amount field separated by tabs. For each name, you want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == "deposits"     { balance[$1] += $2 }
FILENAME == "withdrawals"  { balance[$1] -= $2 }
END                        { for (name in balance)
                                   print name, balance[name]
} ' deposits withdrawals
```

The first statement uses the array **balance** to accumulate the total amount for each name in the file *deposits*. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name is created by the second statement. The **END** action prints each name with its net balance.

13

Random choice

The following function prints (in order) **k** random elements from the first **n** elements of the array **A**. In the program, **k** is the number of entries that still need to be printed, and **n** is the number of elements yet to be examined. The decision of whether to print the **i**th element is determined by the test **rand( ) < k/n**:

```
function choose(A, k, n) {
        for (i = 1; n > 0; i++)
                if (rand( ) < k/n--) {
                        print A[i]
                        k--
                }
}
```

Shell facility

The following **awk** program simulates (crudely) the history facility of the C shell:

```
$1 == "=" {  if (NF == 1)
                        system(x[NR] = x[NR-1])
                else
                    for (i = NR-1; i > 0; i--)
                                if ( match(x[i],$2) ) {
                                        system(x[NR] = x[i])
                                        break
                                }
                    next
           }

$1 != "="      { system(x[NR] = $0) }
```

A line containing only = re-executes the last command executed. A line beginning with =⟨space⟩*cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

Chapter 14

Manipulating text with sed

This chapter describes the stream editor, **sed**(C). It contains information on the following topics:

What is sed?

sed is a tool which allows you to perform large-scale, noninteractive editing tasks. For example, **sed** can replace all the occurrences of a given word in a file with a different word, or delete all lines containing a target string from a file. **sed** is particularly good at working with large files or running complicated sequences of editing commands on a file or group of files from within a shell script; **sed** is usually faster than **ed** at performing global substitutions within a file.

The **sed** program is derived from **ed**, although there are considerable differences between the two, resulting from the different characteristics of interactive and batch operation. **sed** is best thought of as a relative of the **grep** family of commands: just as **grep** is primarily a tool for searching for regular expressions in files, so **sed** is a tool for carrying out regular expression based search and replace operations on files.

sed works on only a few lines of input at a time and does not use temporary files, so the only limit on the size of the files you can process is that both the input and output fit simultaneously on your disk. You can apply multiple "global" editing changes to your text in one pass.

14

Note, however, that **sed** lacks relative addressing (the ability to specify a line number to work on relative to the current line number) because it processes a file one line at a time and never backs up. Also, note that **sed** gives you no immediate verification that a command has altered your text in the way you actually intended. For this reason, you should check your output carefully.

Using sed

sed reads its standard input a line at a time, carries out whatever editing changes are specified, then writes the changed lines to its output. It acts as a filter on the input file, which streams through it.

input file   output file
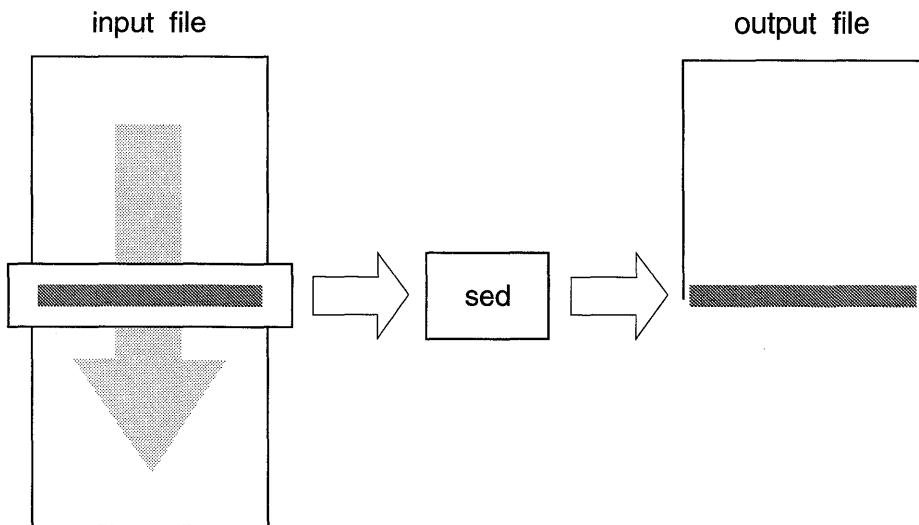
How sed works

To use **sed**, you need to specify the name of the command file that contains your editing script, then the file or files that you are processing. For example:

```
sed -f editscript <inputfile >outputfile
sed -e "/Hello/s/Hello/Hi/g" <letter.old >letter.new
```

Three optional flags are recognized on the command line:

-n   Directs **sed** to copy only those lines specified by **p** functions or **p** flags after **s** functions. See "Whole-line oriented functions" (page 378) for details.

-e   Indicates that the next argument is an editing command.

-f   Indicates that the next argument is the name of the file which contains editing commands, typed one to a line.

Writing sed commands

The general format of a **sed** editing command is:

[*address1*[,*address2*]] *function* [*arguments*]

An address is a parameter which tells **sed** which lines to apply the function to. Addresses may be line numbers or regular expressions. If two addresses are specified, **sed** applies the function to all the lines in the range from *address1* to *address2*. You can omit one or both addresses; in the absence of an address **sed** applies the function to every line that it reads. For example, the following script turns every occurrence of "red" into "blue" throughout a file:

s/red/blue

The function determines the operation that **sed** carries out on the matching line. A function is always required in a command, but arguments are optional for some functions.

Any number of blanks or tabs can separate the addresses from the function, and tab characters and spaces at the beginning of lines are ignored.

How sed commands are carried out

sed commands are applied one at a time in the order they appear (unless you change this order with one of the "flow-of-control" functions discussed in "Functions" (page 377)). **sed** works in two phases, compiling the editing commands in the order they are given, then executing the commands one-by-one on each line of the input file.

When **sed** begins to read the input file, it copies the first line of the file into its "pattern space". The pattern space is an area of memory used by **sed** to store the text which it is currently editing. **sed** then executes its list of commands, applying them to the contents of the pattern space.

sed only carries out those editing functions which correspond to an address matching the current pattern space. For example, if a command is specific to lines 50-150 of a file, **sed** will not carry it out if the pattern space does not contain one of those lines. (**sed** does not print any warnings when this happens: in general **sed** either silently ignores errors, or terminates abruptly with an error message.)

The first matching command is carried out on the text contained in the pattern space; then the second command is carried out on whatever remains in the pattern space, and so on. When no more commands remain to be executed, **sed** appends the contents of the pattern space to its output file and reads another line, then starts processing again, looping back to the first command.

14

Even if you change this default order of applying commands with one of the two flow-of-control functions, **t** and **b**, the input line to any command is still the pattern space resulting from the application of any previously executed commands.

You should also note that the range within which pattern matching occurs is normally one line of input text. If you need to carry out edits that cross line boundaries, you can read more than one line into the pattern space by using the **N** function described in "Multiple input-line functions" (page 384).

The rest of this section discusses the principles of **sed** addressing, followed by a description of **sed** functions.

Addresses

The following rules apply to addressing in **sed**. There are two ways to select the lines in the input file to which editing commands are to be applied: with line numbers or with "context addresses".

Line addresses

A line number is a decimal integer. As **sed** reads lines from its input file, it increments a line counter. Each time **sed** reads a new line it checks whether the current line number matches any of the commands in its command list. If any of the commands match the current line number, they are carried out. The counter runs cumulatively through multiple input files; thus, if **sed** is reading in five files, each 100 lines long, line address 369 actually refers to the 69th line of the fourth file. The counter is not reset when new input files are opened. A special case is the dollar sign character ($) which matches the last line of the last input file.

Context addresses

Context addresses are regular expressions enclosed in slashes " / ". If you specify a context address for a command, **sed** only applies the editing function to those lines which match the regular expression. By using context addresses and a print function, you can improvise a **grep**-like behavior; for example, the shell script *mygrep*:

```
sed -n -e "/$1/p" <$2
```

This script uses the shell parameter $1 as a context address for **sed** to use in searching the file specified by the parameter $2. Whenever **sed** finds a line matching the address given in $1, it executes the **p** function (print) and outputs that line.

Note the **-n** argument to **sed** in this script; **sed** normally echoes every line it reads to its standard output. While the **-n** option is in effect, **sed** only prints when you tell it to with the **p** or **P** functions. Note also that if you want to use **sed** within a shell script and pass parameters to it, the **sed** instructions must be in *double* quotes, not single quotes (see "How the shell works" (page 241) for an explanation of shell quoting and its meaning).

For example:

```
mygrep charlie /etc/passwd
charlie::8:5:Charles Stross:/usr/charlie:/usr/bin/ksh
```

Context addresses are enclosed in slashes (/). They include all the regular expressions common to both **ed** and **sed:**

- An ordinary character is a regular expression and matches itself.

- A caret (^) at the beginning of a regular expression matches the null character at the beginning of a line.

- A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.

- The characters (\n) match an embedded newline character, but not the newline at the end of a pattern space.

- A period (.) matches any character except the terminal newline of the pattern space.

- A regular expression followed by a star (*) matches any number, including 0, of adjacent strings matching the regular expression.

- A string of characters in square brackets ([ ]) matches any character in the string, and no others. If, however, the first character of the string is a caret (^), the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.

- A concatenation of regular expressions is one that matches a particular concatenation of strings.

- A regular expression between the sequences "\(" and "\)" is grouped, and can be referred to as a unit by the **s** function. (Note the following specification.)

- The expression (\d) means the same string of characters matched by an expression enclosed in " \ (" and " \ ) " earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of " \ (", counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.

- The null regular expression standing alone is equivalent to the last regular expression compiled.

14

For a context address to "match" the input, the whole pattern within the address must match some portion of the pattern space. If you want to use one of the special characters literally, that is, to match an occurrence of itself in the input file, precede the character with a backslash (\) in the command.

Each **sed** command can have 0, 1, or 2 addresses.

- A command with no addresses specified is applied to every line in the input. For example:

  s/red/green/

  This command substitutes the first instance of "green" for "red" on all lines.

- A command with one address is applied to all lines that match that address. For example:

  /mike/s/fred/john/

  substitutes the first instance of "john" for "fred" only on those lines containing "mike".

- A command with two addresses is applied to the first line that matches the first address, then to all subsequent lines until a match for the second address has been processed. An attempt is made to match the first address on subsequent lines, and the process is repeated.

  Two addresses are separated by a comma. For example:

  50,100s/fred/john/

  Substitutes the first instance of "john" for "fred" from line 50 to line 100 inclusive. (Note that there should be no space between the second address and the **s** command.)

- If an address is followed by an exclamation mark (!), the command is applied only to lines that do not match the address. For example:

  50,100!s/fred/john/

  substitutes the first instance of "john" for "fred" everywhere except lines 50 to 100 inclusive.

Here are some examples based on the following configuration file (a piece of an */etc/passwd* file):

```
root:x:0:0:Superuser:/:
remacc:x:::Remote access::
daemon:No login:1:1:Spooler:/usr/spool:
sys:No login:2:2:System information::
bin:x:3:3:System administrator:/usr/src:
xmail:x:4:4:Secret Mail:/usr/spool/pubkey:
msgs:No login:7:7:System messages:/usr/msgs:
charlie:x:8:5:Charles Stross:/usr/charlie:/bin/ksh
```

/oo/	matches lines 1, 3, 6 in our sample file
/o*o/	matches lines 1, 3, 4, 6, 7
/[Cc]h.*/	matches line 8
/^o/	matches no lines
/./	matches all lines
/o$/	matches no lines

You can use a single address to control the application of a group of commands by grouping the commands with curly braces ({ }). For example:

/red/ {
s/red/green/
s/blue/yellow/
}

This short **sed** script searches for lines containing the regular expression "red" and then carries out the grouped commands, which replace the first occurrence of "red" with "green" and the first instance of "blue" with "yellow" on each matching line. You might use this script by placing it in a file called *subst.red* and invoking it from a shell as follows:

$ sed -f subst.red <input_file >output_file

For more on substitution in **sed**, including how to apply a change to all instances of a matching string, see "Substitute functions" (page 379).

Functions

All **sed** commands require a function to tell them what to do; the command itself is a combination of an address and a function. Functions are named by a single character. The following types of function are available:

- Whole-line oriented functions that add, delete, and change whole text lines.
- Substitute functions that search and substitute regular expressions within a line.
- Input-output functions that read and write lines and/or files.
- Multiple input-line functions that match patterns that extend across line boundaries.
- Hold and get functions that save and retrieve input text for later use.
- Flow-of-control functions that control the order of application of functions.
- Miscellaneous functions.

14

Whole-line oriented functions

These functions are used to manipulate an entire line at a time in the pattern space, rather than a string within a line. The following whole-line oriented functions are available:

d *Deletes* from the file all lines matched by its addresses. No further functions are executed on a deleted line. As soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line. The maximum number of addresses is two. For example, **/charlie/d** deletes all lines containing the string "charlie".

n Reads the *next* line of input into the pattern space. The current line is sent to the standard output, if appropriate, and the line counter is incremented by one; the execution of editing commands continues following the **n** function rather than looping back up to the top of the command list (as it would if **sed** had naturally exhausted its command list on the current pattern space, and read a new input line). The maximum number of addresses is two.

The following three commands must be specified over multiple lines and the text must appear on a new line. Interior newlines must be hidden by a backslash character (\ ) immediately preceding each newline. The text argument is terminated by the first unhidden newline, which is the first one not immediately preceded by backslash. Once a function executes successfully, the text is written to the output regardless of what later commands do to the line that triggered it, even if the line is subsequently deleted. The text is not scanned for address matches, and no editing commands are attempted on it, nor does it cause any change in the line number counter.

a *Appends* text. The text following the **a** function is appended when the pattern space is sent to the standard output. For example, the following sequence appends the line "(Not to mention blue)" *after* any line containing the word "red":

/red/a
(Not to mention blue)

The **a** command has only one possible address.

i *Inserts* text. When followed by a text argument, **i** functions the same as **a**, except that the text is written to the output *before* the matched line. It has only one possible address. The following sequence inserts the line "(Not to mention blue)" *before* any line containing the word "red":

/red/i
(Not to mention blue)

c *Changes* text. The **c** function deletes the lines selected by its addresses, and replaces them with the lines in the text. (This function is principally used for replacing an entire line with a different line, not for routine find and replace operations (for which the **s**, *substitute*, function is used.) The following example searches for lines containing the word "secret", deletes them, and replaces them with the text "[this line has been censored]":

/secret/c
[this line has been censored]

The **c** function may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of the text is written to the output, not one copy per line deleted. After a line has been deleted by a **c** function, no further commands are attempted on it. If text is appended after a line by an **a** function, and the line is subsequently changed, the text inserted by the **c** function is placed before the text of the **a** function.

As with all **sed** commands, these three multiline commands may be used either in scripts or on the shell command line. In the latter case, the first line's trailing backslash must be quoted, as follows:

$ sed -e "/secret/c
> [this line has been censored]" input_file

Substitute functions

The substitution functions change parts of lines selected by a context search within the line, using the syntax:

[*address*]s/*pattern*/*replacement*/*flags*

The **s** function replaces part of a line selected by the designated *pattern* with the *replacement* pattern. This is similar to **vi** (see "A quick tour of vi" (page 132)). The substitution is restricted to only those lines matching the optional address. The pattern argument contains a *pattern*, exactly like the patterns in addresses. The only difference between a pattern and a context address is that a pattern argument may be delimited by any character other than space or newline. By default, only the first string matched by the pattern is replaced, except when you use the **g** flag.

The replacement argument begins immediately after the second delimiting character of the pattern, and must be followed immediately by another instance of the delimiting character.

14

The replacement is not a pattern, and the characters that are special in patterns do not have special meaning in replacement. Instead, the following characters are special:

  &   This character is replaced by the string that matches the pattern. For example:

s/fruit/fresh & vegetables/

This command substitutes the word "fruit" with "fresh fruit & vegetables".

  \ *d*   *d* is a single digit that is replaced by the *d*th substring matched by parts of the pattern enclosed in " \(" and " \)". If nested substrings occur in the pattern, the *d*th substring is determined by counting opening delimiters.

/\(foo\)\(bar\)/\2\1/

This substitution consists of a regular expression in two groups; "foo" and "bar". If applied to a line containing the word "foobar" the regular expression matches each of the grouped subexpressions in turn. The replacement argument "\2\1" transposes the order of the first two grouped expressions that **sed** matched, producing the following output:

```
barfoo
```

As in patterns, you can make special characters literal by preceding them with a backslash (\ ).

A flag argument can contain the following:

  g   *Global.* Substitutes the replacement for all non-overlapping instances of the pattern in the line. After a successful substitution, the scan for the next instance of the pattern begins just after the end of the inserted characters; characters put into the line from the replacement are not rescanned.

The following example replaces all occurrences of the word "password" with a string of x's:

s/password/xxxxxxx/g

  n   Causes the substitution to be performed only on the *n*th instance of a matching string. For example:

s/Current/Expired/2

This command substitutes the second instance of "Current" in each input line with "Expired". The default value of *n* is 1, which is why a substitution operation specified without the global flag affects only the first instance of a matching string (see the examples in "Context addresses" (page 374)). The maximum value of *n* is 512.

p   Prints the line if a successful replacement was done. The **p** flag causes the line to be written to the output only if a substitution was actually made by the **s** function. Note that if several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line are written to the output (one for each successful substitution). For example, **s/password/xxxxx/gp** globally replaces "password" with x's and prints the results.

w *file*   Writes (appends) the line to a file if a successful replacement was done. The **w** flag causes lines that are actually substituted by the **s** function to be written to the named file. If the filename existed before you run **sed**, it is overwritten; if not, the file is created. A single space must separate **w** and the filename. The possibilities of multiple different copies of one input line being written are the same as for the **p** flag. A combined maximum of ten different filenames can be specified after **w** flags and **w** functions.

The **p** and **w** substitution flags have the same effect as the **p** and **w** functions (see "Input-output functions" (page 382)), with the exception that they are dependent on the specified substitution succeeding.

Here are some examples of the commands in use. When applied to the */etc/passwd* file used previously, **/charlie/s/charlie/charles/w changes** sends the following to standard output:

```
root:x:0:0:Superuser:/:
remacc:x:::Remote access::
daemon:No login:1:1:Spooler:/usr/spool:
sys:No login:2:2:System information::
bin:x:3:3:System administrator:/usr/src:
xmail:x:4:4:Secret Mail:/usr/spool/pubkey:
msgs:No login:7:7:System messages:/usr/msgs:
charles:x:8:5:Charles Stross:/usr/charlie:/bin/ksh
```

At the same time, the following is written to the file *changes*:

```
charles:x:8:5:Charles Stross:/usr/charlie:/bin/ksh
```

(This might be used, for example, to change the login name of the user "charlie" to "charles".)

The following is a shorthand version of the same command:

/charlie/s//charles/w changes

In this case, the context address is the same as the string to be substituted. In this case, **sed** assumes that the value of the null field is the same as that of the preceding (address) field.

The command **s/:/\⟨Tab⟩/gp** (where ⟨Tab⟩ is a typed tab character) replaces all colons with tabs in a data file, and prints the result (for example, as input to an **awk** program).

14

Note that where a substitution command involves a literal backslash, it must be quoted, as follows:

$ **sed -e "s/⟨Tab⟩/\ \ \ \/gp" input_file**

This replaces all the tabs in the input file with a single backslash. Note also the following:

$ **sed -e 's/⟨Tab⟩/\ \/gp' input_file**

This has the same effect. The difference between the two quoting mechanisms is that the double quotes prevent the expansion of all special characters *except* the backslash, the dollar sign and the single quote.

Note that it is not essential that the character used to separate the fields in the substitute command always be a slash. Consider the case where a pathname is to be substituted with another pathname. In such a case, the slashes in the pathname substitution strings conflict with those used to build up the command itself. This can be avoided by using another string as the delimiter character. The following example uses the exclamation mark (!) to specify a substitution command where the affected strings are pathnames:

s!/dev/null!/dev/stdout!gp

The transform function

The transform function (**y**) takes two strings as arguments. It turns each instance of the first character in string 1 into the first character in string 2, then the second character in string 1 into the second character in string 2, and so on. In the following example, "a" is transformed into "A", irrespective of whether it is followed by "b":

y/abcde/ABCDE/

The components of the two strings are literals, and the transformation performed depends on position. As with the substitute command, **y** takes an optional address as an argument:

15y/abcde/ABCDE/

Input-output functions

These functions enable **sed** scripts to read and write files directly:

p   The print function writes the contents of the pattern space (that is, any lines matching the desired address) to the standard output file at the time **sed** encounters the **p** function, regardless of what succeeding editing commands do to the lines. The maximum number of possible addresses is two.

wfile The write function appends the addressed lines to *file*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands do to them. Exactly one space must separate the **w** function and the filename. The combined number of write functions and **w** flags may not exceed 10.

r The read function reads the contents of the named file, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched its address. If **r** and **a** functions are executed on the same line, the text from the **a** functions and the **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and the filename; only one address is possible. If a file mentioned by an **r** function cannot be opened, it is considered a null file rather than an error, and no diagnostic is given.

The **p** and **w** functions perform the same operations as the corresponding substitution flags (see "Substitute functions" (page 379)) with the exception that they are not dependent on a successful substitution.

You may have up to 20 appends in one script, and 10 **w** files open. These limits are built into **sed**.

In the example below, the file *support.extensions* contains the following list of telephone extensions:

```
Fred:   x5706
Marge:  x5631
Sally:  x5239
```

The command **/phone list:/r support.extensions** can be applied to a file like the following:

```
Thank you for enquiring about our customer support hotline.
Please place your calls to our exchange at 346-4573, then
dial one of the following extensions:

phone list:
```

The command has the following effect:

```
Thank you for enquiring about our customer support hotline.
Please place your calls to our exchange at 346-4573, then
dial one of the following extensions:

phone list:
Fred:   x5706
Marge:  x5631
Sally:  x5239
```

14

In this way, **sed** can be used to automatically insert smaller files into the file currently being edited.

Note, however, that the line matching the context address (in the example, the line consisting of the string "phone list:") *must* be terminated with a newline.

Multiple input-line functions

UNIX system pattern-matching operations typically use a single line of input. **grep**, for example, cannot handle embedded newlines. **sed**, however, supplies three uppercase functions that deal specially with multiline pattern spaces.

Within a multiline pattern space, an embedded newline is matched by (\n). The usual end-of-line notation ($) matches only the last newline in the pattern space; preceding embedded newlines are ignored. The start-of-line notation (^) matches the beginning of the pattern space.

N This function appends the next input line to the current line in the pattern space; the resulting lines in the pattern space are separated by an embedded newline. A maximum of two addresses is permitted.

D Deletes from the start of the pattern space all the characters up to and including the first newline character it comes to. If the pattern space becomes empty (the only newline being the terminal newline), another line is read from the input. Following a **D** function the execution of editing commands begins over again from the top of **sed**'s list of commands. **D** takes a maximum of two addresses.

P Prints from the start of the pattern space up to and including the first newline. The maximum number of addresses is two.

If there are no embedded newlines in the pattern space, the **P** and **D** functions are equivalent to their lowercase counterparts.

The multiline functions on their own are not sufficient to match patterns that cross a line boundary. The problem is that embedded newlines may appear anywhere in the pattern space. There are two ways to deal with this: either insert optional newlines between every character in the search string, or strip the \n characters out of the pattern space while searching.

The second method is preferred, but to carry out such a search it is necessary to discard scanned lines on a rolling basis: this requires the ability to make a temporary copy of the pattern space. Techniques for copying the pattern space are described in "Hold and get functions" (this page).

Hold and get functions

In addition to the pattern space, **sed** provides a second buffer called the hold space. The contents of the pattern space can be copied to the hold space, then back again. No operations are performed directly on the hold space. **sed** provides a set of hold and get functions to handle these movements.

h   The **h** (hold) function copies the contents of the pattern space into a holding area, destroying any previous contents of the holding area. The maximum number of addresses is two.

H   The **H** function appends the contents of the pattern space to the contents of the holding area. The former and new contents are separated by a newline.

g   The **g** function copies the contents of the holding area into the pattern space, destroying the previous contents of the pattern space.

G   The **G** function appends the contents of the holding area to the contents of the pattern space. The former and new contents are separated by a newline. The maximum number of addresses is two.

x   The exchange function interchanges the contents of the pattern space and the holding area. The maximum number of addresses is two.

14

Suppose you want to search for a phrase in a file, but are not sure whether there is a newline between two words in the phrase. The logical procedure to search for a phrase straddling two lines is the following:

```
while (input file exists) {
        begin:
        search for phrase in pattern space
        if found
                print
                goto begin
        else
        while (input file exists) {
                append the next line to the pattern space
                save a copy of the pattern space
                discard the first line from the pattern space
                search for phrase in the pattern space
                if found
                        print
                else
                        restore the saved pattern space
                        strip the newline out of the pattern space
                        search for phrase in the pattern space
                        if found
                                print
                        fi
                        discard the first line in the pattern space
                fi
        }
        fi
}
```

The two crucial requirements for this procedure are that it should use a *multi-line* pattern space, adding and deleting lines from it as the script rolls through the file; and that it must save a copy of the pattern space, make destructive changes to the original, and then retrieve the original copy.

The following is a shell script, not a **sed** script. It begins by invoking **sed**: all following commands are enclosed within single quotes. The script accepts two arguments: the string to search for (quoted, if it contains spaces or regular expressions) and the file to search.

```
sed '
/'"$1"'/b
N
h
s/.*\n//
/'"$1"'/b
g
s/ *\n/⟨Space⟩/
/'"$1"'/{
g
b
}
g
D' $2
```

Note that ⟨Space⟩ denotes a literal space character at this point. The first line of **sed** commands searches for the target phrase; if it is present, **sed** branches (and goes back to the beginning of its script). See "Flow-of-control functions" (page 388) for details of the **b** command.

If no match was made, the **N** function appends the next line to the pattern space; the current pattern space is then temporarily saved in the hold space (with the **h** function).

sed now removes the first line from the pattern space, then carries out another search for its target string. (If successful, it loops back to the start of the script.) If it still has not found the target string, it copies the saved version of the pattern space back in again and replaces the newline with a space; it searches for the target string again and, if it finds it, prints both lines.

If **sed** cannot find the target string in its pattern space, it discards the first line in the space and then begins all over again, working from the current line downwards.

Flow-of-control functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part. (They were used in the example in "Hold and get functions" (page 385).)

!   This function causes the next function written on the same line to be applied to only those input lines *not* selected by the address part. There are up to two possible addresses.

{   This function causes the next set of commands to be applied as a block to the input lines selected by the addresses of the grouping function. The first of the commands under control of the grouping function can appear on the same line as the { or on the next line. The group of commands is terminated by a matching } on a line by itself. Groups can be nested and can have up to two addresses.

:label   The label function marks a place in the list of editing functions that can be referred to by **b** and **t** functions. The *label* can be any sequence of eight or fewer characters; if two different colon functions have identical labels, an error message is generated, and no execution attempted.

b*label*   The branch function causes the sequence of editing functions being applied to the current input line to be restarted immediately after encountering a colon function with the same label. If no colon function with the same label can be found after all the editing functions have been compiled, an error message is produced, and no execution is attempted. A **b** function with no label is interpreted as a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line. Two addresses are possible.

t*label*   The **t** function tests whether any substitutions have been made successfully on the current input line. If so, it branches to the label; if not, it does nothing. The flag that indicates that a successful substitution has been executed is reset either by reading a new input line, or by executing a **t** function.

Comments in sed

The # function introduces a comment. Text following a comment is ignored, up to the first newline. If the last character on the comment line is a backslash, the following line is interpreted as a continuation of the comment, up to the first newline:

```
# This sed script transforms source files written \
using the troff mm macro library into troff ms source.
```

This sequence has exactly the same effect as the following:

```
# This sed script transforms source files written
# using the troff mm macro library into troff ms source.
```

If the character immediately following the # character is an "n", no output will be automatically generated by the script. This construction has the same effect as the **-n** command line option (see "Using sed" (page 372)).

Miscellaneous functions

There are three other functions of **sed** not discussed in the sections above.

= The = function writes the number of the line matched by its address to the standard output. One address is possible.

The length of a file can be found using the following command:

$ =

This command is not counting the lines, but rather showing the length of the file by printing the number of the last line.

l The list function is used to display the contents of the pattern space. Non-printable characters such as BEL (\a) are displayed as two digit ASCII codes.

q The **q** function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated. One address is possible. This function could be used to shorten processing if, for example, the objective was to truncate a very long file at the first occurrence of a given word, by copying the input to the output then quitting as soon as the word was encountered.

Appendices

Appendix A

An overview of the system

This appendix introduces the key concepts underlying the SCO OpenServer system. While this material is not essential, it provides a perspective for issues discussed elsewhere in this guide.

The following topics are discussed:

- origins of the UNIX system (this page)
- the design of the UNIX operating system (page 394)
- filesystems and devices (page 403)
- system tools (page 407)

Origins of the UNIX system

The UNIX system evolved over a twenty-year period, and a distinctive philosophy emerged among the programmers who developed it. Unlike previous operating systems, the UNIX system was designed to be *consistent*; while it may initially look complex, once you understand the interrelation between the components, and learn to reason along the same lines as the developers, you will find the UNIX system straightforward to use. This is because the consistency of design was pursued at all levels, and the same general design philosophy was applied to virtually all the components of the system.

The UNIX operating system grew through several generations. The very first implementation of UNIX, by Ken Thompson of Bell Labs in 1970, ran on a DEC PDP-7 minicomputer with only 8K of RAM and a magnetic tape drive.

Today, large UNIX systems run on everything from personal computers to mainframes more than a billion times as powerful as the first system. This highlights the three most important characteristics of the UNIX philosophy:

- scalability

- modularity

- portability

This design philosophy is explained below, with examples of how to apply it to everyday problems in using and understanding the SCO OpenServer system.

The design of the UNIX operating system

The UNIX system is most commonly portrayed as an onion; several layers surround an inner core.
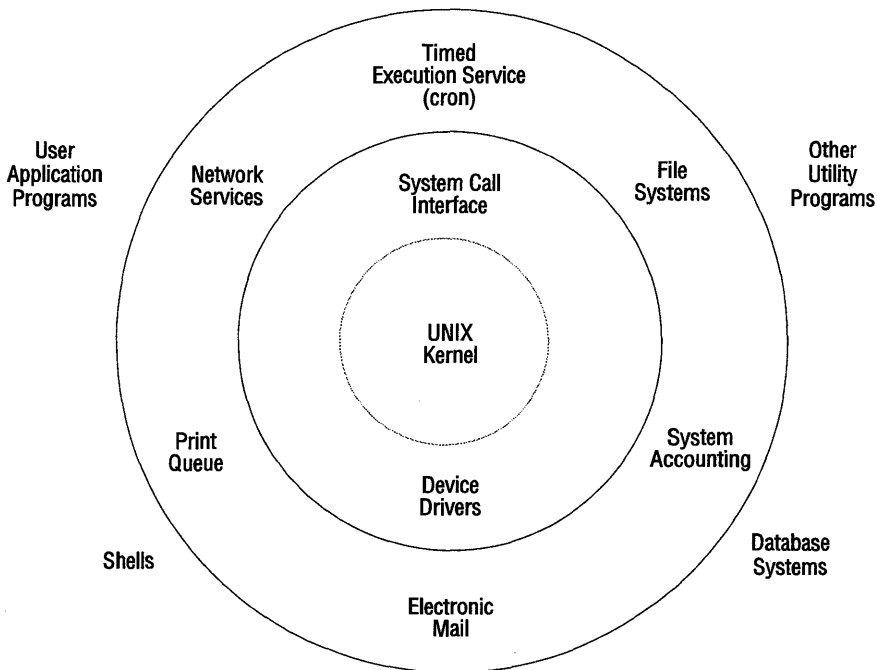
The UNIX operating system

The applications level

The outermost level of the system is the one with which most users interact. This is the applications level. Applications are complex programs designed to automate some business task (such as word processing, spreadsheet calculations or database services). The SCO Shell is an application provided as part of the operating system; other applications are available from third party suppliers. Consult the *SCO Directory* for details of products available for the SCO OpenServer system.

Applications do not exist in a vacuum. Although a user may never see any other part of the system, the spreadsheet or word processor needs a way of storing files, producing a display on the user's terminal, and printing files out (to name just a few of its requirements). For example, the SCO Shell e-mail application uses the **lp** print service to print messages; this saves it from needing a print program of its own. Most applications behave like this, using lower level tools to carry out tasks which would otherwise have to be duplicated.

Applications also use system calls provided by the kernel. The kernel (the core of the UNIX system) provides a uniform interface to all the facilities of the system; this is described in more detail below.

The system utilities

The system utilities lie below the application layer. These are user-accessible programs such as the shells, e-mail delivery agents, and tools such as **awk**, **ls** or **vi**; the tools discussed in this book. You can log in and use these programs without worrying about the intricacies of the underlying system. The tools level provides a set of components which you can use to build special-purpose programs of your own (see Chapter 11, "Automating frequent tasks" (page 245)), or use as simple filters for manipulating files.

Also accessible at this level are a number of user-oriented systems. For example, the (optional) UNIX text processing system appears at this level; so do the tools of the development system (used by C programmers to write applications).

In general, the main difference between a tool and an application is that the application tends to shield you from having to know anything about the UNIX system, whereas to make effective use of software tools, it is necessary to have at least some understanding of the way components of the system fit together.

System services

User level programs (applications and tools) make use of a repertoire of facilities: the system services. These are sets of programs that provide a service essential to the UNIX system, but which are not intended to interact directly with users. For example, the **mmdf** mail system is a system service; it comprises a set of programs designed to deliver e-mail to users automatically. When you use the SCO Shell e-mail application to create a mail message and send it, **mmdf** takes over and determines where the mail is going, then places it in the appropriate spool directory.

Most services are mediated by a daemon. A daemon is a process that runs without human intervention in response to some event. For example, **cron** responds to the system clock and runs processes that are scheduled for a given time.

System services are not part of the kernel, and can be removed or replaced, though doing so may damage or alter the usability of the system. The common UNIX system services include:

- Electronic mail. This is not just the **mail** or SCO Shell **email** program used to read and send mail (the mail delivery agents or MDAs), but the "back end": the programs that spool outgoing mail, work out which host to send it to, dial up the host and transmit the mail to the host's own mail server, then route incoming mail to the correct user's mailbox. (A spooler is a set of service programs that control a queue of files, adding files to it at one end and removing them at the other end for despatch to the destination.) The back end is known as the mail transport agent (or MTA) and is one of the main UNIX subsystems.

- Security and auditing. The first UNIX systems were built on academic research computers; security was not a major concern at first. Later, as the system became a standard for open systems and commercial use, steps were taken to provide tools that allow systems to be secured against accidental damage or deliberate misuse. These subsystems are actually a component of the System Administration tools described below.

- Print services. On a multiuser system, which may be connected to several printers directly or over a network, it is necessary to maintain a queue of print jobs (documents waiting to be printed), which are assigned to printers as they become available. This service is controlled by a print spooler, a set of service programs that maintain the print queue and send files from it to the selected printers.

- Terminal services. The UNIX system can be used with almost any serial terminal, and a wide range of other types of terminal. However, before you can use a text editor like **vi** on a terminal, the UNIX system needs to take control over various aspects of the terminal; cursor positioning, highlighting, and the input and output of characters must all be handled directly. Two services are used by the UNIX system to communicate with character terminals (the "termcap" and "terminfo" databases), and one service (the X protocol) is used to communicate with bitmapped (graphics) terminals.

- Timed execution services. A number of facilities exist that allow the UNIX system to carry out tasks automatically, at a given time. The **cron** daemon keeps a list of jobs to be started at various times and dispatches them when they are due to run. These jobs include not only user programs (like long print jobs) but system tasks (such as flushing the buffer cache, calling another computer to exchange mail files, and backing up filesystems across a network).

The UNIX system kernel

At the center of the UNIX onion is a program called the kernel. Although you are unlikely to deal with the kernel directly, it is absolutely crucial to the operation of the UNIX system. The kernel provides the essential services that make up the heart of UNIX systems; it allocates memory, keeps track of the physical location of files on the computer's hard disks, loads and executes binary programs such as shells, and schedules the task swapping without which UNIX systems would be incapable of doing more than one thing at a time. The kernel accomplishes all these tasks by providing an interface between the other programs running under its control and the physical hardware of the computer; this interface, the system call interface, effectively insulates the other programs on the UNIX system from the complexities of the computer. For example, when a running program needs access to a file, it cannot simply open the file; instead it issues a system call which asks the kernel to open the file. The kernel takes over and handles the request, then notifies the program whether the request succeeded or failed. To read data in from the file takes another system call; the kernel determines whether or not the request is valid, and if it is, the kernel reads the required block of data and passes it back to the program. Unlike DOS (and some other operating systems), UNIX system programs do *not* have access to the physical hardware of the computer. All they see are the kernel services, provided by the system call interface.

The system call interface is an example of an API, or application programming interface. An API is a set of system calls with strictly defined parameters, which allow an application (or other program) to request access to a service; it literally acts as an interface. (For example, a large database system might provide an API that allows programmers to write external programs that request services from the database.)

Kernel sub-processes

Note that the kernel is not indivisible. There are a small number of kernel sub-processes which are executed by the kernel; they are visible in the process listing when you type the command **ps -ef**. Nor is the kernel the first program that runs when you boot (start up) the UNIX system; see "The UNIX system life cycle" (page 400). In addition, the kernel contains device drivers.

A device driver is a sub-program which is designed to enable the kernel to communicate with a peripheral device (such as a hard disk drive, or a local area network adapter) not normally supported by the UNIX system. When the kernel receives a request from a program to read or write to a device which requires a driver, it forwards the request to the device driver to provide the service. Device drivers are linked into the kernel. Some third party add-on components come with their own device drivers; these can be added to the kernel by the administrator.

In addition to device drivers and special processes, the kernel keeps track of the files stored on the system. Files and devices accessed via drivers are referenced through a name space; this is an abstract space in which named objects (the files, directories and devices of the UNIX system) have a uniform interface. The kernel translates references to named objects into requests for actual data stored by means of a protocol called a filesystem or via a device driver. For further details, see "Understanding filesystems and devices" (page 403).

How multi-tasking works

Because the SCO OpenServer system is designed to be open, it is necessary for any application or user-written program running on it to "see" the same environment and be able to call upon the same services, regardless of the specific computer that the UNIX system is running on. To this end, the system provides programs with an API that assumes an almost unlimited extent of memory (up to 4GB or 4096MB per process), and various other facilities; for example, the exclusive use of the computer.

Because contemporary computer technology does not lend itself to creating machines with an unlimited quantity of memory or number of processors, the kernel maps the demands of the application or user program to the resources available.

Several programs may be executed concurrently by scheduling each process to run for a fraction of a second; a round robin arrangement is used so that each process appears to be running continuously and the users see no indication that their instance of **vi**, for example, is spending most of its time in a state of suspension. The kernel keeps track of how much time each process spends running, so that no processes are ignored; however, the more processes are running concurrently, the less time each process spends being processed by the CPU, and the systems appears to run more slowly.

Memory management

Memory is also managed by the kernel. A computer capable of running a multi-tasking system may have to deal with widely varying demands for memory. At some times, no users may be logged on and the only tasks running are the kernel and possibly a backup process. At other times, ten or more users may all be carrying out complex operations which demand lots of memory. The SCO OpenServer system provides "virtual" memory. That is, while each process "sees" an allocation of 4GB of memory available to it, in fact the kernel only doles out memory to a process as and when it requests it: when a process is not actually running, the kernel can swap or page the contents of its physical memory out to a swap space on disk, freeing it up temporarily for another process. The user's process refers to each unit of memory by its address (a number indicating where in the address space of the process, that is, its available memory, the unit lies), and the kernel maps from the requested address to the real memory address, which may be located somewhere else in the computer's memory or paged out to disk. Disk accesses are very slow compared with memory accesses. Consequently, a process that demands a lot of memory may spend much of its time waiting for pages of memory to be read in or written out to disk.

When the system runs low on memory, it may forcibly swap out processes that have been running for a while to make room for other processes. When memory demands exceed the memory available, it may begin to "thrash;" that is, the time it spends swapping processes in and out may vastly exceed the amount of time it spends running them.

The advantage of virtual memory is that, while it is not inexhaustible, it does not run out suddenly; the system does not abruptly refuse to start new processes because it is running low on memory. Thus, as the load on the system increases, its performance tends to degrade gradually. (The system may refuse to run more processes if its process table becomes full. This is extremely unlikely to happen in normal use.)

Two processes are spawned by the kernel to handle paging and swapping. Paging refers to the allocation of virtual memory. "Pages" of memory are mapped onto the processes' address space, and the kernel process keeps track of whether the page is physically present in the computer's memory or is present on the hard disk. Swapping is a mechanism which allows an entire process to be dumped out of memory and stored on a swap partition on the hard disk. These processes are **sched** and **vhand**, processes 0 and 2 respectively. You cannot kill these processes and you will never see executable files called *sched* or *vhand* on the filesystem because they are parts of the kernel.

The UNIX system life cycle

Unlike a small computer operating system, the UNIX system is designed to run continuously. Constant operation allows it to schedule tasks a long way in advance, and ensures that the services it provides are available on demand, whenever they are needed. All systems need to be shut down periodically for maintenance, but it is not uncommon for a SCO OpenServer system to run for several weeks or even months between shutdowns.

Whenever you start up a UNIX system, it goes through the following complex life cycle:

Startup   When you switch a personal computer on, it is not yet ready to run the UNIX system. The machine contains a library of programs stored in read-only memory (or ROM) which are known collectively as the BIOS (or basic input-output system). These programs serve two purposes; they are used by DOS (but not the UNIX system) to access peripheral devices, and they carry out a power-on self-test (or POST) of the computer's hardware.

If all is well, the light on the computer's first floppy disk drive flashes. If a disk containing a small program called a boot program is present in the drive, it will then read the program in and proceed to the next stage; if there is a hardware fault and the POST fails, the computer will either beep a series of tones at you or display a message on its monitor, and refuse to go any further.

Boot   If the power-on self-test was successful, the boot process commences. The term "boot" is a traditional reference to the way in which the system must pull itself up by its own bootstraps, loading a short program that runs and loads a more elaborate program, and so on. The first step in the boot process is for the computer to read in a short program stored in the first sector of either a floppy disk or the first hard drive on the computer. This is carried out by part of the BIOS. The boot sector program then loads another short program, which is just intelligent enough to search the disk for a program called **/boot**, and read that program into the computer's main memory.

At this stage, the computer is minimally functional. The **/boot** program cannot make use of virtual memory, mount filesystems, or do any of the other tasks associated with the system; neither can it run under the system when it is operational. What it *can* do is prompt you for the name of a file to execute, then search the *root* directory of the *root* filesystem for that file and load it. To do this it places a prompt on the system console:

```
Boot:
```

and then it waits for you to type the name of the kernel, or any additional instructions that it recognizes. If you do not type anything, it will time out after a specified period and load the default file listed in */etc/default/boot*.

Loading the kernel

The kernel is visible on your system as a file in the root directory, usually called */unix*. The **boot** program copies the kernel into the computer's memory, then starts it running.

When the kernel begins to run, it starts by setting up a number of internal lists, or tables. These tables are used to keep track of running processes, memory allocation, open files, and a number of other things; they are not directly accessible to you. However, two of them which are of interest are the process table (portions of which you can list out with the **ps** command) and the buffer cache, which is described in "Understanding filesystems and devices" (page 403).

After initializing its tables, the kernel creates three dummy processes; **sched**, **vhand** and **bdflush** (with process IDs 0, 2 and 3 respectively). These processes are sections of kernel code which must be called periodically; **vhand** provides virtual memory paging services, **sched** provides swapping services, and **bdflush** flushes the buffer cache periodically. None of these processes can be killed; they are part of the kernel, and are essential to the correct running of the UNIX system.

Finally the kernel creates a third process; **init**, or process 1. **init** starts up as a dummy process, then achieves independence: it runs as the first true process on the system. **init** runs continuously; it is the parent of all other processes on the system.

Run levels

> Run levels define the behavior of **init**, and by extension those processes which run on the system when it is at any given level. The system starts at run level 0 (shutdown) and then enters run level 1, single user mode. At level 1, only the root filesystem is mounted and only processes connected to the console can run; this means that it is safe to check the unmounted filesystems for integrity without risking any other processes altering them. At other run levels, **init** starts up the daemon processes that provide various services, and enters multiuser mode.
>
> **init** executes other programs via the **fork** system call. Each time **init** calls **fork**, it passes control to the kernel, which creates a new entry in the process table, allocates a temporary storage area called a U-area, and copies the calling processes' local data (including the stack) into the U-area. The kernel then returns control to the child process, which may make an **exec** call, overwriting itself with a new program. **init** periodically reads a file called */etc/inittab*, which tells it which programs to execute at any given run level.
>
> The **init**(M) program should not be confused with the **init** process; the former is an executable program which can be used by the administrator to change the run level of the system or cause the **init** process to reread the */etc/inittab* file.

Multiuser mode

> When the system reaches a suitable run level (2 or higher), **init** starts a series of processes called **gettys**. (In */etc/inittab* each line specifying a **getty** process includes the option **respawn**. This means that whenever a **getty** process dies, **init** spawns a replacement with the same parameters immediately. The **getty** options include a *tty* serial line, which it serves.
>
> The job of the **getty** process is to display a login prompt, then wait for input. If you try to log in on the terminal, **getty** works out your serial line speed as you type your name, then **execs** a **login** process. The **login** process reads your password; if it is incorrect the process dies, and **init** spawns another **getty** on your terminal. If you enter a valid password, **login** then runs a shell, which inherits your terminal, your identity and your access permissions.
>
> When your login shell finishes, there are no processes left attached to your terminal, so **init** respawns another **getty** to wait for a login. Thus, while the system is in multiuser mode, each terminal continually executes a four stage cycle: **init-getty-login-shell**.

A

It is not uncommon for a system to remain in multiuser mode for days or weeks at a time. However, it is necessary for the system administrator to shut it down for maintenance at regular intervals. (Performance is likely to suffer if a system is kept running for several weeks without a shutdown.)

Shutdown

When the administrator shuts down a system, it follows a set procedure. Because the system may be supporting a number of users, warning messages are broadcast to all terminals before a shutdown. After a short time, **init** switches to run level 1, killing all the processes linked to terminals and flushing and unmounting all the mounted filesystems. The buffer cache is then flushed (by the **sync** program), and the system drops to run level 0, or shutdown.

It is important to understand that unless you are running on a multiprocessor system only one process is actually being executed at any given instant. Although the system is multitasking, a single processor can only carry out one instruction at a time.

The kernel effectively mediates the demands of each process, by scheduling the processes to run one after another. The signal for the kernel to take over is sent by the system clock; every hundredth of a second the kernel wakes up and checks to see if the current process has had its time allocation. If so, the kernel suspends the process and switches execution to the process on the queue with the highest priority.

The kernel also mediates all requests for memory and requests to load and run other processes. The requesting process (be it **init** or any other process) issues a "system call", a request to the kernel to deliver a service; it then suspends execution (sleeps) until the kernel can deliver the requested facility.

Understanding filesystems and devices

As mentioned above, the UNIX system provides access to information by mapping it within a notional name space. A name space is simply an abstract space within which all entities are identified by name; items existing in the system name space are files of data (including directories), and special files (such as devices) which provide access to hardware devices such as tapes, terminals or hard disks. Given the name of an entity, the kernel can retrieve it and read its associated data. However, the fact that the entities can all be referred to by the same method should not be confused with equivalence; devices, although they look like files, are not files.

Files and filesystems

At the most fundamental level, a file in the UNIX system is a collection of zero or more bytes of information, which can be referred to by name. Files are used to impose a partitioning strategy on the information stored by the computer; in general, the contents of a file relate to a single program, a single database, or a single document, which can then be referred to by name. It is possible to impose an arbitrary structure on the contents of a file, but the file is essentially the main, atomic unit of information in the filesystem.

Files are themselves partitioned by directories. A directory is simply a file that contains a list of other files, and some other information that indicates where exactly the files are stored.

At the physical level, a hard disk drive bears little resemblance to a UNIX system file hierarchy. Data is stored as magnetic field patterns on the surfaces of a set of spinning platters; read/write heads (similar to those of a tape recorder) move across the platters and read the magnetic field patterns, converting them into a stream of bytes which is then fed to the system. The surfaces of the disks are divided into concentric tracks and radial sectors within each track; the same track on each platter of a multi-platter hard disk is called a cylinder.

(The same terms are applied to floppy disks, although they have only two sides. Tapes, on the other hand, are divided up into blocks, running lengthwise along the tape, which correspond to sectors on a disk.)

The smallest unit of data that a hard disk can read is a single sector of a given track. Each sector stores a fixed number of bytes, usually in the range 512 to 8192. Therefore, at some stage, the UNIX system must be able to work out from a given filename on which tracks and sectors the data within a file is stored at, and retrieve that data. There is no direct mapping between the filename and the physical location of its data on a disk.

Because files can be stored on a variety of media, it is necessary for the system to provide a uniform method for referring to files, and this is the purpose of the filesystem.

The term "filesystem" is used in two contexts. In the first it indicates a hierarchy of directories and files on a disk, which is "mounted" on (connected to) another filesystem so that it appears as a subdirectory of the first filesystem. (The directory where the filesystem is mounted is called the mount point.) In the second context, it is a more abstract term; a filesystem is a system for mapping from the name of a file to the physical location of its data on a mountable medium.

In general, a filesystem consists of three components: a superblock, an inode table, and a series of uniquely numbered blocks (corresponding to the sectors on the hard disk). The superblock is the first component of a filesystem. It contains information about the type of the filesystem, its structure, and its size, including where the inode table is, and how many data blocks there are.

Inodes

The inode table starts immediately after the superblock. It contains a fixed number of inodes (pronounced "i-node"), which are data records identified only by number; each inode contains information about permissions, owner-ship, type, the number of bytes in the file, and a number of slots which point to data blocks in the filesystem. Every file on the system has an inode which is associated with it: given an inode number it is possible to retrieve all the data blocks associated with that inode, simply by looking at the data block entries in the inode's record. If the file the inode defines is very big, the inode may contain pointers to an extension block, which contains more slots identifying the components of the file. If the file is huge, the extension block may point to further extension blocks that identify its data blocks.

The first inode in the inode table corresponds to the root directory. A directory is simply a file that contains a list of filenames and their associated inodes. Thus, when you give the system the name of a file you want to access, it looks in the directory to identify its corresponding inode, then reads the inode to identify the data blocks it needs to retrieve. If the file is large, it looks in the extension blocks to find the other blocks it needs.

To locate a file in another directory, the system looks up each directory in turn, identifies the inode of the subdirectory, then looks in that subdirectory until it finds the file. If a directory file is so large that some of its data blocks are referenced indirectly through an extension block, it may take longer to retrieve the inode number of the file; therefore it is desirable to keep the number of entries in a directory file to less than 640 files (if filenames are less than 12 characters long). (Note that when you delete a file, all that happens is that you erase the inode number associated with its name in the current directory. The contents of the inode are not cleared, because the inode may be referred to by another name, or link, in another directory. The inode contains a count of the number of links to it. When the last link is destroyed, the inode is added to a list of free nodes and can be reused.)

Caching

To speed access to the filesystem, the kernel maintains a buffer cache in memory. (The speed with which the computer can read its memory is of the order of a million times as fast as the speed at which it can retrieve data from a disk.)

The cache contains the contents of the disk blocks that have been read from or written to most recently. Whenever a block is read, it is stored in the cache, because the most recently read blocks are also those which are most likely to be read or written to next. Every few minutes, the system purges the cache, writing any recently changed buffers to the disk; alternatively, if a large number of writes accumulate, filling the cache, it may force a purge.

In general the buffer cache significantly improves the performance of the UNIX system, but there is a cost. Because some of the recent writes to the filesystem are stored in memory rather than written straight out to the disk, a power failure or crash can result in the filesystem being corrupted: that is, the inodes may not contain the correct data blocks for their files, the list of free inodes may be incorrect, and the data stored in the most recently written files may also be incorrect. This is why it is vital to follow a careful shutdown procedure and not simply switch the computer off while it is running the SCO OpenServer system.

Because the operating system has evolved over time, it is capable of supporting a number of different types of filesystem. The main differences between them are their speed, efficiency, size of data blocks, capacity, and history; in general it is sufficient to stick to the standard EAFS filesystem which offers long filenames and symbolic links, but the earlier systems are provided to maintain compatibility with older software installations. The system can also support the DOS filesystem structure, which is less efficient than the standard ones. (DOS does not support multiple names for files, long filenames, or a buffer cache.)

It is not uncommon for a UNIX system to have several filesystems mounted on it at once. To understand how this works, you need to understand device files.

Device files

As mentioned above, within the UNIX system name space, files and devices look similar. For example, the commands **cat /dev/tty01** and **cat myfile** both produce a stream of output. However, there is a difference between these two commands; their apparent equivalence is the result of some intricate work on the part of the operating system.

A device file is a special type of file. Rather than pointing to an inode which points to some data blocks, a device file points to an inode that contains some associated information: a major device number (which defines the type of device it is connected to), and a minor device number (which identifies a particular device of that type). When you carry out a file operation on a device file, the system uses the major number to determine which device driver to use to read data from or write data to the device. (The minor number is used internally by the device driver.)

Device files are created with **mknod** and cannot be manipulated like ordinary files, although you can rename them or create links to them.

Device files are typically kept in the directory */dev*. They include identifiers that can be used to read from and write to kernel memory (*/dev/kmem*), hard disk drives (in "raw" or "block" mode, for example */dev/rhd00* for raw access to drive unit 0), and all the terminals, floppy disk drives, tape drives, and other components of the computer. Block devices write through the cache, providing fast, high level access. Raw device files bypass the buffer cache but are more flexible. Raw devices are therefore sometimes used by special applications like databases, which maintain their own high performance cache for the hardware they use for data storage.

This has some useful applications. You can send messages to a terminal by redirecting the output of a **cat** command to that terminal's device file. Alternatively, you can mount a filesystem on a subdirectory of your root filesystem, transparently adding another disk drive to the system. All you need do is create an empty subdirectory, then issue the **mount** command with the device file that refers to the additional filesystem as one of the parameters. The new filesystem is then invisibly attached to the root filesystem at the mount point you created.

As noted above, links between files are simply filenames that share the same inode. Inodes are only unique within a given filesystem. It is therefore impossible for a normal link to cross a filesystem boundary. However, a symbolic link can be used instead. A symbolic link is created using the -s flag to **ln**; instead of pointing to the inode of the file, the symbolic link points to a short file containing a reference to the filesystem and inode of the linked file.

While device files may seem somewhat obscure, they are one of the most important features of the operating system because they allow you to extend it. Your system administrator can add components to the computer, then create a device file through which they can be accessed. This is one of the ways in which the UNIX system is uniquely scalable.

How to think about system tools

Having toured the operating system at the service level, you can now see more clearly how the tools it provides are able to work together. Because all files and devices are equivalent, it is possible to provide software tools that have a uniform interface. The generic UNIX system tools see any file or device as a stream of bytes coming from a standard input. Their function is simple; they carry out some transformation on the stream of bytes, and send the results to an output stream which may be another file or a device like a terminal.

One of the main design goals of the UNIX system was to enforce this equivalence. A secondary goal was to promote flexibility. There are no obvious limits to the ways in which tools can be connected via pipes and shell scripts. It is possible in principle to take the output from any program and feed it to the same program as input; it might not be sensible to do so, but it is at least practical. (Under some other operating systems which enforce rules governing file types, it is impossible to do this.) It is therefore possible to construct self-modifying programs, or arbitrarily complex programs operating repeatedly on the same data files, from first principles using the tools provided with the system.

In addition to having standardized inputs and outputs, the software tools follow a design philosophy. Each tool was originally designed to do one task, and to do it as efficiently as possible. This is still visible in the default behavior of many of the programs. For example, **grep** is designed to search for regular expressions (patterns) in text. Its behavior can be modified by various flags, but in principle it will perform adequately with no arguments other than a pattern to search for and a stream to read its input from. Consequently, there are a number of lowest common denominator standards that ensure that almost all UNIX-like operating systems can be used to run the same shell scripts, as long as they are written with the standards in mind and do not use the specific value-added features of the operating system tools.

Some of the tasks carried out by the system tools are quite complex. A lot of research into the theory of computer languages went into the early development of the UNIX system at the Bell Telephone Laboratories and various universities. Consequently, a number of "power tools" are provided which, on any other system, would be considered to be programming languages in their own right. Other tools are unambiguously recognized as languages. To a large extent, these tools share a common core syntax that demonstrates their common ancestry. If you learn the C programming language, you will see great similarities with the C shell and **awk**; if you take care to learn the regular expression syntax recognized by **egrep** and the search and replace operations of **vi** you will have little difficulty generalizing them to basic commands in **sed** and pattern matching commands in **awk**.

While the body of knowledge you need to master is not small, there are similarities between apparently different tools. After a while, you will be able to learn new system utilities by analogy with those which you are already familiar with. This is the basis of a comprehensive understanding of the SCO OpenServer system.

vi commands

The following tables contain all the basic **vi** commands.

Starting vi

Command	Description
vi *file*	start at line 1 of *file*
vi +n *file*	start at line *n* of *file*
vi + *file*	start at last line of *file*
vi +/*pattern file*	start at *pattern* in *file*
vi -r *file*	recover *file* after a system crash

Saving files and quitting vi

Command	Description
:e *file*	edit *file* (save current file with :**w** first)
:w	save (write out) the file being edited
:w *file*	save as *file*
:w! *file*	save as an existing *file*
:q	quit **vi**
:wq	save the file and quit **vi**
:x	save the file if it has changed and quit **vi**
:q!	quit **vi** without saving changes

Moving the cursor

Keys pressed	Effect
h	left one character
l or ⟨Space⟩	right one character
k	up one line
j or ⟨Enter⟩	down one line
b	left one word
w	right one word
(	start of sentence
)	end of sentence
{	start of paragraph
}	end of paragraph
1G	top of file
nG	line n
G	end of file
⟨Ctrl⟩W	first character of insertion
⟨Ctrl⟩U	up ½ screen
⟨Ctrl⟩D	down ½ screen
⟨Ctrl⟩B	up one screen
⟨Ctrl⟩F	down one screen

Inserting text

Keys pressed	Text inserted
a	after the cursor
A	after last character on the line
i	before the cursor
I	before first character on the line
o	open line below current line
O	open line above current line

Changing and replacing text

Keys pressed	Text changed or replaced
cw	word
3cw	three words
cc	current line
5cc	five lines
r	current character only
R	current character and those to its right
s	current character
S	current line
~	switch between lowercase and uppercase

Deleting text

Keys pressed	Text deleted
x	character under cursor
12x	12 characters
X	character to left of cursor
dw	word
3dw	three words
d0	to beginning of line
d$	to end of line
dd	current line
5dd	five lines
d{	to beginning of paragraph
d}	to end of paragraph
:1,. d	to beginning of file
:.,$ d	to end of file
:1,$ d	whole file

Using markers and buffers

Command	Description
mf	set marker named " f "
`f	go to marker " f "
'f	go to start of line containing marker " f "
"s12yy	copy 12 lines into buffer " s "
"ty}	copy text from cursor to end of paragraph into buffer " t "
"ly1G	copy text from cursor to top of file into buffer " l "
"kd`f	cut text from cursor up to marker " f " into buffer " k "
"kp	paste buffer " k " into text

Searching for text

Search	Finds
/and	next occurrence of "and", for example, "and", "stand", "grand"
?and	previous occurrence of "and"
/^The	next line that starts with "The", for example, "The", "Then", "There"
/^The\>	next line that starts with the word "The"
/end$	next line that ends with "end"
/[bB]ox	next occurrence of "box" or "Box"
n	repeat the most recent search, in the same direction
N	repeat the most recent search, in the opposite direction

Searching for and replacing text

Command	Description
:s/pear/peach/g	replace all occurrences of "pear" with "peach" on current line
:/orange/s//lemon/g	change all occurrences of "orange" into "lemon" on next line containing "orange"
:.,$/\<file/directory/g	replace all words starting with "file" by "directory" on every line from current line onward, for example, "filename" becomes "directoryname"
:g/one/s//1/g	replace every occurrence of "one" with 1, for example, "oneself" becomes "1self", "someone" becomes "some1"

Matching patterns of text

Expression	Matches
.	any single character
*	zero or more of the previous expression
.*	zero or more arbitrary characters
\<	beginning of a word
\>	end of a word
\	quote a special character
*	the character " * "
^	beginning of a line
$	end of a line
[*set*]	one character from a set of characters
[XYZ]	one of the characters " X ", " Y ", or " Z "
[[:upper:]][[:lower:]]*	one uppercase character followed by any number of lowercase characters
[^*set*]	one character not from a set of characters
[^XYZ[:digit:]]	any character except " X ", " Y ", " Z ", or a numeric digit

Options to the :set command

Option	Effect
all	list settings of all options
ignorecase	ignore case in searches
list	display ⟨Tab⟩ and end-of-line characters
mesg	display messages sent to your terminal
nowrapscan	prevent searches from wrapping round the end or beginning of a file
number	display line numbers
report=5	warn if five or more lines are changed by command
term=ansi	set terminal type to "ansi"
terse	shorten error messages
warn	display "[No write since last change]" on shell escape if file has not been saved

Appendix C

DOS command equivalents

This appendix contains a table showing some common MS-DOS commands and their SCO OpenServer system equivalents.

For more information about any of the SCO OpenServer system commands, consult the *Operating System User's Reference*.

The commands listed below are for working with SCO OpenServer system files. If you have DOS installed on the same machine as your SCO OpenServer system, you can access your DOS files from within the SCO OpenServer system. For more information about accessing DOS files, see Chapter 6, "Working with DOS" (page 173) and **doscmd**(C) in the *Operating System User's Reference*.

DOS command	What it does	UNIX system equivalent	Notes
cd	change directories	**cd**(C)	
cls	clear the screen	**clear**(C)	
copy	copy files	**cp**(C), **copy**(C), **tar**(C)	Use **cp** to copy files, **copy** to copy directories, and **tar** to copy files or directories onto floppy disks or tapes.
date	display the system date and time	**date**(C), **cal**(C)	On the UNIX system, **date** displays the date and the time. **cal** displays the date, the time, and a 3-month calendar.
del	delete a file	**rm**(C)	Be careful when using **rm** with wildcard characters, like **rm** *.
dir	list the contents of a directory	**ls**(C)	There are a variety of options to **ls** including **ls -l** to see a long listing, **ls -c** to see a listing in columns, and **ls -f** to see a listing that indicates file types.
diskcomp	make a track-by-track comparison of two floppy disks	**diskcmp**(C)	
diskcopy	copy a source disk to a target disk	**diskcp**(C)	
edlin	line editor	**ed**(C), **ex**(C), **vi**(C)	**vi** is a full-screen text editor with powerful search and replace functions. **ed** and **ex** are predecessors of **vi**.
fc	compare two files	**diff**(C), **diff3**(C), **cmp**(C)	**diff** compares two text files. **diff3** compares three text files. Use **cmp** to compare binary files.
find	find text within a file	**grep**(C)	**grep** (global regular expression parser) finds text within a file. The UNIX system's **find**(C) command finds files.

(Continued on next page)

(Continued)

DOS command	What it does	UNIX system equivalent	Notes
format	format a disk	**format**(C)	See */etc/default/format* for the default drive to format. The **format** command formats a disk for use with UNIX system files. Use **dosformat** (see **doscmd**(C)) to format a DOS disk.
mkdir	make a directory	**mkdir**(C)	
more	display output one screen at a time	**more**(C)	
print	print files in the background	**lp**(C)	Use **lp** *filename* **&** to print in the background. You can run any UNIX system command in the background by adding **&** (ampersand) to the end of the command line.
ren	rename a file	**mv**(C)	
rmdir	remove an empty directory	**rmdir**(C)	Use **rm -r** to remove a directory that is not empty.
sort	sort data	**sort**(C)	
type	display a text file	**cat**(C), **more**(C)	
xcopy	copy directories	**copy**(C), **tar**(C)	Use **tar** if you want to copy directories onto disk or tape.

Appendix D

Sample shell startup files

This appendix contains sample listings and line-by-line explanations of the following shell startup files:

Bourne shell (**sh**)	*.profile*
Korn shell (**ksh**)	*.profile*
	.kshrc
C shell (**csh**)	*.login*
	.cshrc

Line numbers have been added to all the file listings for purposes of explanation; numbers do not appear in the actual files.

The Bourne shell .profile

The Bourne shell (**sh**) reads a single file in your home directory, the *.profile*. A typical Bourne shell *.profile* might look something like this:

```
1  :
2  #    @(#) profile 23.1 91/04/03
3  #
4  # .profile   -- Commands executed by a login Bourne shell
5  #
6  # Copyright (c) 1985-1995 The Santa Cruz Operation, Inc.
7  # All rights reserved.
8  #
9  # This Module contains Proprietary Information of the Santa Cruz
10 # Operation, Inc., and should be treated as Confidential.
11 #
```

```
12 PATH=$PATH:$HOME/bin:.                    # set command search path
13 MAIL=/usr/spool/mail/`logname`           # mailbox location
14 export PATH MAIL
15 # use default system file creation mask
16 eval `tset -m ansi:ansi -m :?${TERM:-ansi} -r -s -Q`
```

line 1   Contains a single colon that says "execute this script as a Bourne
                 shell script." This is a convention for scripts written in Bourne
                 shell, so C shells know to start a new **sh** to run the Bourne shell
                 scripts.

                 (C shells need to start Bourne shells to run Bourne shell scripts
                 because they do not understand the Bourne shell language. The
                 Korn shell, however, is compatible with the Bourne shell, so you
                 can use most Bourne shell scripts in the Korn shell without a
                 problem.)

lines 2-11   Contain comments. Each line that starts with a number sign (#)
                 is a comment. The shell ignores these lines. In this case, lines
                 2-11 contain SCO copyright information.

line 12   Sets the path. It says, "set the path equal to the current path,
                 plus the *bin* in the home directory, plus the current directory (.)."
                 Setting the path to the existing path presumes there is a system-
                 wide */etc/profile* that sets up a path definition for all users. The
                 path definition in */etc/profile* would contain the usual command
                 directories, such as */bin* and */usr/bin*.

line 13   Tells the shell where to find mail. The `logname` in backquotes
                 tells the shell to substitute the output of the command
                 logname(C), which returns a user's login name. Because `log-
                 name` is used instead of a particular login name, this script works
                 for any user.

line 14   Tells the shell to export the **PATH** and **MAIL** settings to all its
                 subshells. This guarantees that if you type **sh** to start a new
                 Bourne shell, the new Bourne shell has the same path definition
                 and mail setup as your login Bourne shell.

line 15   Contains a comment, like lines 2-11. This comment tells us that
                 login Bourne shells use the default system file creation mask,
                 which is set in */etc/profile*. This explains why there is no **umask**
                 setting in this *.profile*.

line 16   Sets up the terminal type, using the **tset**(C) (terminal setup) command. **tset** sets your terminal type, as well as the erase and kill characters for your terminal.

This **tset** command says "check if this serial line is mapped to *ansi* in the */etc/ttytype* file; if it is, set the terminal type to *ansi*. Otherwise, prompt the user with TERM:ansi." The **-r** option prints the terminal type on the screen, **-s** exports the terminal type to any subshells, and **-Q** suppresses the Erase set to . . ., Kill set to . . . messages that **tset** would otherwise show. The **tset** command is enclosed in backquotes and preceded by the shell command **eval** to guarantee that all necessary substitutions are made within the **tset** command before it is evaluated by the shell.

The Korn shell .profile and .kshrc

The Korn shell uses two startup files, the *.profile* and the *.kshrc*. The *.profile* is read once, by your login **ksh**, while the *.kshrc* is read by each new **ksh**.

A typical Korn shell *.profile* might look something like this:

```
1  :
2  #    @(#) profile 23.1 91/04/03
3  #
4  # .profile   -- Commands executed by a login Korn shell
5  #
6  # Copyright (c) 1990-1995 The Santa Cruz Operation, Inc.
7  # All rights reserved.
8  #
9  # This Module contains Proprietary Information of the Santa Cruz
10 # Operation, Inc., and should be treated as Confidential.
11 #
12 PATH=$PATH:$HOME/bin:.                    # set command search path
13 export PATH
14 if [ -z "$LOGNAME" ]; then
15     LOGNAME=`logname`              # name of user who logged in
16     export LOGNAME
17 fi
```

D

```
18 MAIL=/usr/spool/mail/$LOGNAME                # mailbox location
19 export MAIL
20 if [ -z "$PWD" ]; then
21      PWD=$HOME                      # assumes initial cwd is HOME
22      export PWD
23 fi
24 if [ -f $HOME/.kshrc -a -r $HOME/.kshrc ]; then
25      ENV=$HOME/.kshrc              # set ENV if there is an rc file
26      export ENV
27 fi
28 # use default system file creation mask (umask)
29 eval `tset -m ansi:ansi -m $TERM:?${TERM:-ansi} -r -s -Q`
30 # If job control is enabled, set the suspend character to ^Z (control-z):
31 case $- in
32 *m*) stty susp '^z'
33      ;;
34 esac
35 set -o ignoreeof              # don't let control-d logout
36 case $LOGNAME in             # include command number in prompt
37 root)       PS1="!# " ;;
38 *)    PS1="!$ " ;;
39 esac
40 export PS1
41 /tcb/bin/prwarn              # issue a warning if password due to expire
```

line 1   Contains a single colon that says "this is a Bourne shell script."
                Even though this is a startup script for the Korn shell, the
                authors have chosen to use the more common syntax of the
                Bourne shell programming language. This single colon com-
                mand is more portable than the (preferred) newer hash-bang
                syntax. It is equivalent in function to the line:

                #! /bin/sh

lines 2-11   Contain comments.

line 12   Sets the path definition in exactly the same way as the preceding
                Bourne shell *.profile*: "set the path equal to the current path, the
                bin in the home directory, and the current directory."

line 13   Exports the path to any subshells. This way, you do not have to
                include a path definition in your *.kshrc*.

lines 14-17   Set up a variable called **LOGNAME**, which is used in the following **MAIL** setting (line 18). Literally, these lines say "if checking for the value of **LOGNAME** returns a zero-length string (that is, if **LOGNAME** is not set), then set **LOGNAME** to the output of the **logname** command. Then, export the **LOGNAME** variable to all subshells."

line 18   Tells the shell where to look for mail, using the variable **LOGNAME**.

line 19   Exports the mail location to all subshells.

lines 20-23   Check to see if a variable is already set, and if it is not, set the variable. These lines are similar to lines 14-17. In this case, **PWD** is being set to the home directory.

lines 24-27   Check for a *.kshrc* file in the home directory, and set the **ksh** variable **ENV** to this file if it exists. **ksh** looks in the file pointed to by the **ENV** variable to set up the environment for every new **ksh**; you need to tell it explicitly to look in *˜/.kshrc* for **ENV** definitions. Literally, these lines say "if a file called *.kshrc* exists in the home directory and the file is readable, then set **ENV** to point to this file, and export **ENV**."

line 28   Contains a comment. Just as in the preceding Bourne shell *.profile*, **umask** is not set here. The authors have chosen to use the default system **umask** rather than resetting it on a per-user basis.

line 29   Sets up the terminal type using **tset**(C), as explained in the preceding Bourne shell *.profile*.

lines 30-34   Test to see if job control is enabled, and if it is, set the suspend character to ⟨Ctrl⟩**Z**. Job control is a Korn shell feature that lets you move jobs you are processing from the foreground to the background and vice versa. You use the suspend character to suspend a job temporarily that is running in the background.

line 35   Tells the shell to ignore single end-of-file (EOF) characters. This is what you set to stop ⟨Ctrl⟩**D** from logging you out.

lines 36-40   Set up the prompt based on the value of **LOGNAME**. For normal users, the prompt is the current command number followed by a " $ "; for *root* (the superuser), the prompt is the current command number followed by a " # ".

line 41   Runs the command **prwarn**(C), which warns you if your password is due to expire soon.

A typical *.kshrc* might look like this:

```
1  :
2  #
3  # .kshrc      -- Commands executed by each Korn shell at startup
4  #
5  #    @(#) kshrc 1.1 90/03/13
6  #
7  # Copyright (c) 1990-1995 The Santa Cruz Operation, Inc.
8  # All rights reserved.
9  #
10 # This Module contains Proprietary Information of the Santa Cruz
11 # Operation, Inc., and should be treated as Confidential.
12 #
13 # If there is no VISUAL or EDITOR to deduce the desired edit
14 #  mode from, assume vi(C)-style command line editing.
15 if [ -z "$VISUAL" -a -z "$EDITOR" ]; then
16      set -o vi
17 fi
```

line 1   Tells the shell that this is a Bourne shell script by starting the script with a single colon, as you have seen before.

lines 2-14   Contain comments. These make up the bulk of this brief *.kshrc*.

lines 15-17   Set up **vi**(C) as the default editor **ksh** uses when you want to edit a command line. Literally, these lines say "If the **VISUAL** variable is not set, and the **EDITOR** variable is not set, then turn on (**set -o**) the **vi** option."

The C-shell .login and .cshrc

The C shell, like the Korn shell, uses one file to set up the login environment and a different file to set up environments for every subsequent C shell. In C shell, *.login* is the file read only at login, and *.cshrc* is the file read each time a **csh** is started.

While both the Bourne shell and the Korn shell use Bourne shell startup scripts, the C shell uses C-shell startup scripts, so you will notice that variables are set and tests are performed slightly differently. C-shell scripts do not start with a " : " because they are intended for use with C shells, not Bourne shells.

A typical C-shell *.login* might look something like this:

```
1  #    @(#) login 23.1 91/04/03
2  #
3  # .login    -- Commands executed only by a login C-shell
4  #
5  # Copyright (c) 1985-1995 The Santa Cruz Operation, Inc.
6  # All rights reserved.
7  #
8  # This Module contains Proprietary Information of the Santa Cruz
9  # Operation, Inc., and should be treated as Confidential.
10 #
11 setenv SHELL /bin/csh
12 set ignoreeof                              # don't let control-d logout
13 set path = ($path $home/bin .)       # execution search path
14 set noglob
15 set term = (`tset -m ansi:ansi -m :?ansi -r -S -Q`)
16 if ( $status == 0 ) then
17     setenv TERM "$term"
18 endif
19 unset term noglob
20 /tcb/bin/prwarn          # issue a warning if password due to expire
```

lines 1-10   Contain comments.

line 11   Sets the environment variable **SHELL** to be */bin/csh*.

line 12   Tells **csh** to ignore single end-of-file (EOF) characters; in other words, do not let ⟨Ctrl⟩**D** log out, as the comment says.

line 13   Sets the path definition in the same way as the preceding Bourne shell *.profile*: "set the path equal to the current path, the *bin* in the home directory, and the current directory."

line 14   Turns on the **noglob** setting. The **noglob** setting, which prevents filename expansion, is turned on before a **tset**(C) command is attempted (on line 15). Without **noglob**, the **tset** command would be read incorrectly.

lines 15-19   Set up your terminal type using **tset**. Line 15 is the **tset** command you have seen before. Line 16 tests to make sure the **tset** command succeeded and, if it did, line 17 sets the environment variable **TERM**. Line 18 closes the **if** statement. Line 19 unsets the **term** variable and turns off **noglob**, so filenames now expand as expected when wildcard characters are used.

line 20   Runs **prwarn** to warn you if your password is due to expire.

A typical *.cshrc* might look like this:

```
1   #
2   # .cshrc    -- Commands executed by the C-shell each time it runs
3   #
4   #    @(#) cshrc 3.1 89/06/02
5   #
6   # Copyright (c) 1985-1995 The Santa Cruz Operation, Inc.
7   # All rights reserved.
8   #
9   # This Module contains Proprietary Information of the Santa Cruz
10  # Operation, Inc., and should be treated as Confidential.
11  #

12  set noclobber                          # don't allow '>' to overwrite
13  set history=20                         # save last 20 commands
14  if ($?prompt) then
15      set prompt=\!%\            # set prompt string
16  # some BSD lookalikes that maintain a directory stack
17      if (! $?_d) set _d = ()
18      alias   popd    'cd $_d[1]; echo ${_d[1]}:; shift _d'
19      alias   pushd   'set _d = (`pwd` $_d); cd \!*'
20      alias   swapd   'set _d = ($_d[2] $_d[1] $_d[3-])'
21      alias   flipd   'pushd .; swapd ; popd'
22  endif
23  alias print 'pr -n \!:* | lp'          # print command alias
```

lines 1-11   Contain comments.

line 12   Turns on **noclobber,** which prevents you from unintentionally overwriting files using output redirection.

line 13   Sets the length of the command history to 20 commands. Both **ksh** and **csh** keep track of old commands and allow you to re-use them.

lines 14-15 Check to see if the prompt string is set, and, if it is not, set it to be a " % ".

lines 16-22 Set up some command aliases to perform directory stack manipulation. These commands are familiar to users of the Berkeley (Berkeley Standard Distribution — BSD) UNIX system.

line 23   Sets up the **print** alias, which runs files through the **pr**(C) print program before sending them to the printer.

Appendix E
Further reading

This book is too short to do more than provide an introduction to the field of UNIX system programming. Many of the tools described here are complex enough to warrant one or more books to themselves. Although further information is available in the *Operating System User's Reference* you may wish to look elsewhere for advanced tuition on the programming tools covered in this book.

Learning awk

The first place to look for detailed information about the **awk** programming language is *The AWK Programming Language* (Alfred V. Aho, Brian W. Kernighan and Peter J. Weinberger; Addison-Wesley, 1988). This is the "canonical" (definitive) book on **awk**; the authors are the original developers of the language. It contains a tutorial, a detailed definition of the language, and a series of applications ranging from the simple to the highly technical.

A second book on **awk** is *sed & awk* (Dale Dougherty; O'Reilly & Associates, Inc., 1991). This book contains an overview and tutorial in the use of each of these programs.

Learning sed

The canonical reference to **sed** is a technical paper, *sed — a Non-interactive Text Editor*, by L. E. McMahon, reprinted in Volume 2 of the *UNIX Research System Manual*, tenth edition, (Saunders College Publishing, 1992).

As the preceding paper is rather dry, you may prefer to consult the book *sed & awk* (described above).

E

Learning the shells

As each of the shells has achieved prominence, so a definitive book has been published that describes them. These books are not necessarily the most readable ones on the subject, but they are the most authoritative (because at the very least they were co-written by the developer of the shell).

For the Bourne shell, the book is *The UNIX System V Environment* (S. R. Bourne; Addison-Wesley, 1987). This book covers the Bourne shell programming language and provides an introduction to the other software tools which are used in conjunction with it.

For the Korn shell, the book is *The Korn Shell command and programming language* (Morris I. Bolsky and David G. Korn; Prentice-Hall, 1989). This is the standard reference to the Korn shell, and contains a tutorial and detailed description of the programming language supported by the shell.

Another useful book on the Korn shell is *Learning the Korn Shell* (Bill Rosenblatt; O'Reilly & Associates, Inc., 1993).

Learning the C programming language

The C programming language has not been described in this manual. C is provided as part of the SCO OpenServer Development System, and is most useful to professional programmers wishing to write high-performance software. Nevertheless, C is of vital importance to the programs described in this manual. The UNIX system has traditionally been written in C, and the shells and **awk** embody many of the constructs of the C programming language; **awk** in particular resembles a special-purpose subset of C that has been tuned for the purpose of text scanning. The influence of the C programming language is pervasive, and a lot of the superficially peculiar features of the UNIX system make much more sense when viewed with a working knowledge of this language.

Readers who wish to explore the UNIX architecture thoroughly or become programmers may want to learn C. The traditional starting place is the "white book," *The C Programming Language* (Dennis Kernighan and Brian Ritchie; Second Edition, Prentice-Hall Software Series, 1988). The first edition of this book documents the original "K&R" dialect of the C programming language; the second edition has been updated to include the ANSI extensions to the language. The book contains a tutorial in C, followed by a detailed reference, a brief discussion of the UNIX system interface, and the standard library.

There are innumerable other books about C, but this one has the advantage of having been written by the author of the language; it is generally regarded as the standard text on the subject.

While these books explain individual systems in detail, they do not in general attempt to provide an understanding of the philosophy of the system, including an explanation of how the UNIX system was designed and integrated.

There are two approaches to understanding UNIX, besides understanding it as a user. The first approach is the system level approach, which attempts to explain the system in terms of the services it provides to applications, and covers the API (Application Program Interface) of the operating system in some depth. This is most useful to programmers wishing to develop applications that can take full advantage of the UNIX environment.

A useful example of this type of book is *Advanced UNIX Programming* (Marc Rochkind; Prentice-Hall, 1985). This book presupposes a familiarity with the C programming language. On that basis, it conducts the reader on a guided tour of the intricacy of UNIX system programming, with a chapter by chapter overview of the function calls available to user programs. There are many other books of this type; this one was one of the first detailed explanations of UNIX system programming.

The second approach to understanding the UNIX system is the internals approach, which provides the reader with a detailed explanation of how the internal subsystems of the UNIX operating system were designed, and how they carry out their functions. This course of study almost certainly requires a basic knowledge of operating systems theory and computer science before it can be made use of, but provides the suitably equipped reader with a total understanding of what the UNIX operating system was designed to achieve, and how it succeeds.

The classic text following this approach is *The Design of the UNIX Operating System* (Maurice J. Bach; Prentice Hall, 1986). Bach provides a detailed exposition of the design elements of the UNIX system kernel, including information on how processes are scheduled, how memory is managed, and how the API is presented to the applications run on the system.

For a more introductory text (but one for which a knowledge of the C programming language is still required), see *Operating Systems: Principles and Practice* (Andrew S. Tannenbaum; Prentice-Hall). This book does not cover the UNIX system as such, but it provides a first undergraduate course in the theory and practice of operating systems from a very UNIX system-like perspective. The centerpiece of the book is the kernel of an operating system called Minix, which is a small system developed for teaching purposes. The principles explained in this book are almost entirely applicable to the world of UNIX systems, and it has the advantage (for those who are new to operating systems theory) of explaining the central concepts as it goes along.

absolute mode

A method of changing file permissions using 3-digit octal numbers. For example, to add group write permission on a file called *report* using absolute mode, type **chmod 664 report**. Note that you must be *root* or the owner of the file to change permissions on that file. You can also change permissions using symbolic mode.

absolute pathname

A pathname for a file or directory that begins at the *root* directory. Every absolute pathname begins with a slash character (/), which stands for the *root* directory. See also pathname and relative pathname.

application

A computer program that performs a particular task. Word processing, spreadsheet, and database programs are all applications. See also Applications list.

Application folder

A sublist on the main Applications list, which usually includes a list of related application programs. An Application folder can contain applications and other application folders. See also Applications list.

Applications list

The list of available applications and application folders that is displayed on the main SCO Shell screen. See also application and Application folder.

archive

To place a file or group of files in a form convenient for storage on backup media such as floppy disks or tape. Normally, you archive backup files or files that are important but not often used. Such files can be copied to and from backup media using the **Archive** option on the **File** menu. See also backup.

argument

A word you type on the command line that is separated by a space from the command itself. A command can have more than one argument. Arguments tell a command how to you want it to work. For example, **lf -a**; the **-a** option tells the **lf** file listing program that you want it to show *all* files. These types of arguments are also known as options or flags. Arguments can also tell a command what you want it to work on: for example, **lf -a /tmp/spell.out** tells **lf** to list the file */tmp/spell.out* if it exists.

ASCII

The American Standard Code for Information Interchange is a standard way of representing characters on many computer systems. The term "ASCII file" is often used as a synonym for "plain text file," that is, a file without any special formatting, which can be viewed using UNIX system utilities such as **cat**(C), **more**(C), and **vi**(C).

attribute

Attribute bits are set on a file to control which users have permission to read, write, or execute it. See permissions.

backup

A copy of one or more files, directories, or filesystems that is stored apart from the original to safeguard against unplanned deletion. Used as a verb, it means to create a backup copy. The **Archive** option on the **File** menu allows you to copy files to backup media for safekeeping. See also archive.

Bourne shell

A UNIX system shell, named after its author, Steven R. Bourne. To start a Bourne shell from the command line, type **sh** and press ⟨Enter⟩.

buffer

An area of computer memory used to store information temporarily before it is written out to a more permanent location, like a file.

C shell

An alternative UNIX System V shell supplied with the SCO OpenServer system. This shell, written by William Joy at the University of California at Berkeley, is known for its interactive features, such as the ability to recall and modify previous command lines. The C shell shell programming language has a syntax like that of the C language, hence the name. C shell is the standard shell on older versions of the Berkeley UNIX operating system found at many universities. To start a C shell from the command line, type **csh** and press ⟨Enter⟩.

command alias

An alternative name for a command. When you type the alias, the command is substituted for the alias. Aliases are useful when you remember commands by names other than their UNIX system names; for example, DOS users may think of **dir** rather than **ls** when they want to list a directory. Aliases are also useful for creating commands that perform several UNIX system commands at once. See the *Operating System User's Guide* for more information.

command line

The instructions you type next to the shell prompt. Command lines can contain commands, arguments, and filenames. You can enter more than one command on a command line by joining commands with a pipe (|), or by separating commands using the command separator (;). The shell executes your command line when you press ⟨Enter⟩.

command separator

The semicolon (;) serves as a command separator on the UNIX system. If you want to issue several commands on one line, separate the commands with semicolons before you press ⟨Enter⟩. For example, type **ls; pwd** and press ⟨Enter⟩ to list files and then print the working directory. Commands are executed in sequence as separate processes.

current directory

See current working directory.

current working directory

The directory where you are currently located. Use the **pwd**(C) command (print working directory) to see your current working directory. The current working directory is taken as the starting point for all relative pathnames. This directory is symbolically referred to as "." in directory listings.

device

Peripheral hardware attached to the computer such as a printer, modem, disk or tape drive, terminal, and so on. Devices in the SCO OpenServer system are controlled by device drivers which are linked into the kernel.

directory

Where the UNIX system stores files. Directories in the UNIX system are arranged in an upside-down tree hierarchy, with the root (/) directory at the top. All other directories branch out from the root directory. The UNIX system implements directories as normal files that store the names of the files within them.

environment

The various settings that control the way you work on the UNIX system. These settings are specific to the shell you use and can be modified from the command line or by modifying shell control files. For example, the directories the shell searches to find a command you type are set in the variable **PATH**, which is part of your environment.

environment variable

Special variables that modify your login shell behavior. Typical examples are **PATH**, which defines the directories in which the shell will search for files or commands, and **PROMPT** which determines the on-screen shell prompt message. See also variable.

error message

A message informing you that the computer cannot perform the task you requested. The error message briefly describes the nature of the problem.

file

The basic unit of information on a UNIX filesystem. Regular files are usually either text (ASCII) or executable programs. Other types of files exist on the UNIX system such as directories, which store information about the files within them; device files, which are used by the system to access a particular device; and FIFO (First In First Out) pipe files, which are used to transfer data between programs. The attributes of each file are stored in the file's inode.

See also directory.

file descriptor

A number associated with an open file; used to refer to the open file in I/O redirection operations.

full pathname

See also absolute pathname.

group

A set of users who are identified with a particular group ID number on the UNIX system. Typically, members of a group are coworkers in a department or on a project. Each file on the UNIX system also has a group associated with it; this group, along with the owner and the permissions controls who can access and modify that file. You can see the group of a file by listing the file with the l command. To find out your own group, use the **id**(C) command.

home directory

The place in the filesystem where you can keep your personal files and sub-directories. When you log in, you are automatically placed in your home directory. Typically, this will be */u/loginname* or */usr/loginname*, where *loginname* is your login name. The shell's shorthand for the home directory is " ~ ". See tilde expansion.

inode

The internal representation of a file, showing disk layout, owner, type (see file), permissions, access and modification times, size and the number of links. Each inode has a unique decimal identifier.

kernel

The central part of the UNIX operating system, which manages how memory is used, how tasks are scheduled, how devices are accessed, and how file information is stored and updated.

Korn shell

Written by David Korn, it is compatible with the Bourne shell, but provides a much wider range of programming features. The Korn shell also offers improved versions of many of the C shell's interactive features. To start a Korn shell from the command line, type **ksh** and press ⟨Enter⟩. See also Bourne shell and C shell.

link

A filename that points to another file. Links let you access a single file from multiple directories without storing multiple copies of the file. If you make a change to the content of a linked file, the change is reflected in each of the links. All links point to an inode. See also symbolic link.

literal

A literal character or string is one that represents itself, that is, that can be taken literally (as opposed to a pattern, that represents some other characters). For a metacharacter to regain its literal value (for example, for * to mean an asterisk and not "zero or more characters") it must be "quoted". See quoting and wildcard.

log in

The way you gain access to a UNIX system. To log in, you enter your login name and password and the computer verifies these against its user account records before allowing you access.

log out

What you do after you have finished working on a UNIX system. You can log out by pressing ⟨Ctrl⟩**D**, typing **exit**, or typing **logout**, depending on your shell.

login name

The name through which you gain access to the operating system. When you are logging onto the computer, you must enter this login name, followed by a password.

login shell

The shell that is automatically started for you when you log in. You can start to work in other shells, but your login shell will always exist until you log out.

macro

A collection of instructions or keystrokes that may be invoked using a single name or keystroke combination, used to automate regular and complex tasks.

mail alias

A single name used to send mail to several users at once. For example, many users have aliases set up for mailing to the entire company, single departments, or groups of individuals.

manual page

An entry in a UNIX reference manual. These entries can be accessed online using the **man**(C) command. A letter in parentheses following a command or filename refers to the reference manual section where the command or file is documented. For example, the **man**(C) command is documented in section C, Commands, of the *Operating System User's Reference*. They are also called "man pages."

mask

A series of bit settings that "cover up" existing settings, only allowing some settings to show through, while masking out others.

metacharacter

A special character that is replaced by matching character strings when interpreted by the shell. Metacharacters, which define the form of a string, and literal characters, which match only themselves, make up regular expressions.

multitasking

A system that can do several jobs at once.

multiuser

A system that can be used by more than one person at the same time.

named buffer

A buffer used to copy text between files in the **vi**(C) editor. **vi** clears unnamed buffers when it switches files, but the contents of named buffers are preserved.

online

Accessible from your terminal screen.

operating system

A group of programs that provide basic functionality on a computer. These programs operate your computer hardware in response to commands like **copy** and **print**, and form a set of functional building blocks upon which other programs depend. An operating system also manages computer resources such as peripheral devices like disk drives or printers attached to the computer and resolves resource conflicts, as when two programs want to use a disk drive at the same time.

owner

1. The user who created a file or directory. Only the owner and *root* can change the permissions assigned to the file or directory.

2. One of the attributes of a file that, along with its group and permissions, determine who can access and modify that file. You can see the owner of a file by listing it with the l command. Use the **chown**(C) command to change the owner of a file.

password

The string of characters you are prompted for after you type your login name when you are logging in. Your password is the key that lets you into the UNIX system; you should choose it wisely, keep it secret, and change it regularly. Use the **passwd**(C) command to change your password.

path

The directory list through which your shell searches to find the commands you type. Your path is stored in the shell variable **PATH**.

pathname

The name of a directory or a file, for example, */usr/spool/mail*. Each component of a pathname, as separated by slashes, is a directory, except for the last component of a pathname, which can be either a directory or a file. A single word by itself, such as *Tutorial*, can be a pathname; this is a relative pathname for the file or directory *Tutorial* from the current working directory. A single slash, (/), is the pathname for the root directory. See also absolute pathname and relative pathname.

permissions

The settings (also called properties or attributes) associated with each file or directory that determine who can access or modify the file and directory. Use the 1 command to list a file's permissions; use the **chmod**(C) (change mode) command to change a file's permissions.

pipe

A way of joining commands on the command line so that the output of one command provides the input for the next. To use a pipe on the command line, join commands with the vertical bar symbol, (|). For example, to sort a file, eliminate duplicate lines, and print it, you could type **sort** *file* | **uniq** | **lp**.

print job

A request you have made to the printer to print a file. Each print job has an ID number that you can see using the **lpstat**(C) command. You can cancel a print job by typing **cancel** and its job ID number, then pressing ⟨Enter⟩.

process ID

A number that uniquely identifies a running program on the UNIX system. This is also known as the PID.

prompt

One or more characters or symbols that identify a line on which commands can be entered, as in a UNIX or DOS window. "Prompt" also refers to the text displayed when the computer displays a request for input or an instruction. The default prompt can be replaced by setting the PS1 environment variable.

quoting

A mechanism that is used to control the substitution of special characters. Special characters enclosed in single quotes are not replaced by their meaning, but remain embedded in the text when the quotes are stripped off. Double quotes are used to prevent the expansion of all special characters except " $ ", " \ " and " ` ".

regular expression

A notation for matching any sequence of characters. The notation is used to describe the *form* of a sequence of characters, rather than the characters themselves. Regular expressions consist of literal characters, which match only themselves, and metacharacters.

relative pathname

A pathname that does not start with a slash (/); for example; *Tutorial*, *Reports/September*, or *../tmp*. A relative pathname is searched for, starting from the current working directory and may use the notation " .. " to indicate "one directory up from the current working directory." See also absolute pathname and pathname.

root

The top directory of a UNIX filesystem, represented as a slash (/). Also, the login name of the superuser, a user who has the widest form of computer privileges.

shell

A program that controls how the user interacts with the operating system. Using such programs, you can write a shell script to automate work you do regularly. The shells available with the SCO OpenServer system include the Korn shell, the Bourne shell, and the C shell.

shell escape

A command you type from within an interactive program to escape to the shell. In **vi**, you can type **:!***command* to escape to the shell and execute *command*. When *command* has finished executing, you are returned to the editor. You can start a new shell this way with **:!sh**, for example. To exit this subshell and return to the editor, press ⟨Ctrl⟩**D** or type **exit**.

shell programming language

A programming language that is built into the shell. The Korn shell, the Bourne shell, and the C shell all have slightly different programming languages but all three shells offer basics such as variable creation, loops, and conditional tests.

shell script

An executable text file written in a shell programming language. Scripts are made up of shell programming commands mixed with regular UNIX system commands. To run a shell script, you can change its permissions to make it an executable file, or you can use it as the argument to a shell command line (for example, **sh** *script*). The shell running the script will read it one line at a time and perform the requested commands.

shell variable

A variable associated with a shell script.

standard error

The usual place where a program writes its error messages. By default, this is the screen. Standard error can be redirected; to a file, for example. Also known as stderr.

standard input

The usual place from which a program takes its input. By default, this is the keyboard. Standard input can be redirected; for example, you can use the less-than symbol (<) to instruct a program to take input from a file. Also known as stdin, the standard input is identified by the file descriptor 0.

standard output

The usual place where a program writes its output. By default, this is the screen. Standard output can be redirected; for example, you can use a pipe symbol (|) to instruct a program to write its output into a pipe, which will then be read as input by the next program in the pipeline. Also known as stdout, the standard output is identified by the file descriptor 1.

subdirectory

A directory that resides within another directory. Every directory except the *root* directory is a subdirectory.

superuser

A user who has powerful special privileges needed to help administer and maintain the system. The superuser logs in as *root*. Someone with the superuser or *root* password can access and modify any file on the system.

symbolic link

A new name that refers to a directory or file that already exists. Use this name to change to another directory without typing its full pathname. Unlike normal links, symbolic links can cross filesystems and link to directories. See also link.

symbolic mode

A method of changing file permissions using keyletters to specify which set of permissions to change and how to change them. For example, to add group write permission on a file called *report* using symbolic mode, you could type **chmod g+w report**. Note that you must be the owner of a file or the superuser to change permissions on that file. You can also change permissions using absolute mode.

system administrator

The person who looks after the day-to-day running of the computer and performs tasks such as setting up user accounts and making system backups.

terminal

Video display unit with a keyboard, a monitor, and sometimes a mouse. They do not have any independent processing power themselves and they must be connected to a computer before they can do any useful work.

terminal type

A name for the kind of terminal from which you are working. Typically, the terminal type is an abbreviation of the make and model of the terminal, such as *wy60*, which is the terminal type for a Wyse60. Your terminal type is stored in the variable **TERM**.

tilde expansion

The ability of the shell to translate instances of the tilde character (~) into the pathname of the user's home directory.

trash directory

A subdirectory of a user's home directory in which files are temporarily stored before being removed from the system permanently. The trash directory is represented on screen by the Trash icon. You can remove files by dropping them on the Trash icon.

umask

A permissions mask that controls the permissions assigned to new files you create. You can set your umask from the command line or in one of your shell startup files.

user account

The records a UNIX system keeps for each user on the system.

variable

An object known to your shell that stores a particular value. The value of a variable can be changed either from inside a program or from the command line. Each shell variable controls a particular aspect of your working environment on the UNIX system. For example, the variable **PS1** stores your primary prompt string.

wastebasket

A directory where deleted files are temporarily stored. Once you delete a file, it remains in the wastebasket directory until you exit SCO Shell, at which point all files in the wastebasket are permanently deleted. You can recover files from the wastebasket (provided you have not exited SCO Shell) using the Wastebasket option on the File menu. See also trash directory.

wildcard

A character (such as " ? " or " ∗ ") that is substituted with another character or a group of characters in text searches and similar operations. See also metacharacter.

Symbols, numbers

A

Index

E

F

Q

R

Index

U

V

W

X

1 May 1995

AU20003P001