SERIES 60 (LEVEL 68)
MULTICS STORAGE SYSTEM
PROGRAM LOGIC MANUAL
ADDENDUM A

RESTRICTED DISTRIBUTION

SUBJECT

Description of the Multics Storage System

SPECIAL INSTRUCTIONS

This is the first addendum to the Program Logic Manual (PLM) that describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the *Multics Programmers' Manual, Commands and Active Functions* (Order No. AG92), *Subroutines* (Order No. AG93), and *Subsystem Writers' Guide* (Order No. AK92).

Change bars indicate where technical changes have been made. Appendix A is new and does not contain change bars. The changes contained in this addendum will be incorporated into the next revision of the manual.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 6.0

ORDER NUMBER

AN61A, Rev. 0

September 1978

27275
2.5C480
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through vi	iii through vi
1-1, 1-2	1-1, 1-2
2-5, 2-6	2-5, 2-6
2-9, 2-10	2-9, 2-10
2-21, 2-22	2-21, 2-22
3-1 through 3-8	3-1 through 3-8
4-3, 4-4	4-3, 4-4
4-7 through 4-10	4-7 through 4-10
4-15 through 4-18	4-15 through 4-18
4-21, 4-22	4-21, 4-22
4-25, 4-26	4-25, 4-26
5-3 through 5-6	5-3 through 5-6
5-9, 5-10	5-9, 5-10
5-13, 5-14	5-13, 5-14
6-3 through 6-14	6-3 through 6-10
	6-10.1, blank
	6-11 through 6-14
6-17 through 6-20	6-17 through 6-20
6-23, 6-24	6-23, 6-24
7-1 through 7-4	7-1 through 7-4
9-13, 9-14	9-13, 9-14
16-1, 16-2	16-1, 16-2
16-5 through 16-8	16-5 through 16-8
17-5 through 17-8	17-5 through 17-8
	A-1 through A-19, blank

SERIES 60 (LEVEL 68)

MULTICS STORAGE SYSTEM PROGRAM LOGIC MANUAL

RESTRICTED DISTRIBUTION

SUBJECT

Description of the Multics Storage System

SPECIAL INSTRUCTIONS

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the *Multics Programmers' Manual, Commands and Active Functions* (Order No. AG92), *Subroutines* (Order No. AG93), and *Subsystem Writers' Guide* (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which complete, will supersede the *System Programmers' Supplement to the Multics Programmers' Manual* (Order No. AK96).

SOFTWARE SUPPORTED

Multics Software Release 5.0

The information contained in this document is the exclusive property of Honeywell Information Systems. Distribution is limited to Honeywell employees and certain users authorized to receive copies. This document shall not be reproduced or its contents disclosed to others in whole or in part.

ORDER NUMBER

AN61, Rev. 0

July 1977

Honeywell

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS K-28)
Phoenix, Arizona 85005

This PLM explains and describes the subsystems and data bases involved in the reader's understanding of the organization, goals, and design of the software involved. This is not to say that explanations as detailed and thorough as in more traditional PLMs do not appear. However, these discussions are not intended to be read unless all of the Sections preceding these discussions have been understood. It is hoped that the reader will appreciate this approach.

This Program Logic Manual (PLM) describes the internal organization of those parts of the Multics supervisor responsible for implementing the Multics virtual memory. This information is accurate as of Multics Release 5.0. The subsystems described by this document are commonly known as page control, segment control, and volume management.

This PLM assumes familiarity with the overall functional organization of the Multics Operating System, and the user interface as presented in the Multics Programmers' Manual, Order Nos. AG91, AG92, AG93, AK92, AX49. Some familiarity with the Honeywell 68/80 processor is assumed.

Other relevant Program Logic Manuals are:

Order No	Name
AN71	Reconfiguration
AN70	System Initialization

CONTENTS

Page

Section 1 Introduction. 1-1

Section 2 Segment Control Overview and Concepts 2-1
VTOC, and Disk-resident Segment Images . . . 2-1
Activation Information. 2-4
File Map. 2-7
Permanent Information 2-7
Active and Nonactive Segments. 2-9
VTOC Attributes. 2-10
AST Hash Table and Determining Activity. . . 2-10
AST Hierarchy. 2-10
Breakdown of the AST Entry 2-11
AST Lists and Threads. 2-17
AST Replacement Algorithm. 2-18
AST Trickle. 2-19
Locking Conventions. 2-19
Trailers and Setfaults 2-21
Boundsfaults 2-22
Segment Moving 2-22
Encacheability Control 2-22

Section 3 The VTOC Manager. 3-1
Introduction and Overview. 3-1
General Policies 3-2
VTOC Buffer Segment. 3-3
Description of the VTOC Buffer Control Word, vtoc_buffer.b. 3-4
Organization of the VTOC Manager 3-5
VTOC Buffer Replacement Strategy 3-7
Error Strategy 3-8
ESD Strategy 3-8
VTOCE Allocation/Deallocation Service of VTOC Manager. 3-9
Services of VTOC Manager for Demounting. . . 3-9

Section 4 Services of Segment Control 4-1
Creation of Segments 4-2
Physical Volume Selection Algorithm . . . 4-3
Deletion of Segments 4-4
Segment Truncation 4-5
Satisfying Segment Faults. 4-6
Significance of +activate+. 4-7
Segment Fault Handler. 4-8
Activation 4-12
Deactivation 4-13
VTOCE Updating 4-14
Descriptor Segment Management. 4-17
Boundsfault Handling 4-18
Setting and Reporting on VTOC Attributes . . 4-20
PDS and KST Management 4-21

		Page
	Semi-Permanent Activation (grab_aste). . . .	4-23
	IOI and FNP6600 Buffer Segment Special-Casing.	4-25
	Segment Moving	4-25
	Special Services for sweep_pv.	4-29
	Services on Behalf of the Hierarchy Salvager	4-31
	Demand Deactivation of Segments.	4-34
	Services at Demount/Shutdown Time.	4-34
Section 5	Page Control Overview and Concepts.	5-1
	Basic Goals and Services of Page Control . .	5-2
	Basic Organization of Page Control	5-4
	Page Table Lock.	5-6
	Outline of the Data Bases of Page Control. .	5-6
	Zero Pages	5-9
	Main Memory Replacement Algorithm.	5-9
	Paging Device Management Algorithm (Page Multilevel)	5-12
Section 6	Page Control Data Bases	6-1
	Page Control Device (devadd)	6-1
	Paging Data Objects.	6-4
	PTW, or Page Table Word.	6-4
	Core Map	6-7
	Core Map Entry (CME)	6-7
	Paging Device Map.	6-10
	Paging Device Map Entry (PDME)	6-10.1
	PDMAP Header	6-14
	PVTE Variables for Page Control.	6-15
	Synopsis of Relevant SST Variables	6-16
Section 7	Address Management Policy	7-1
	Introduction and Nulled Address.	7-1
	Implications of Finite Packs	7-4
	Non Segment-Movability of the Supervisor . .	7-5
	Guaranteed Bootability of the Supervisor . .	7-5
	RPV Parasite Segments.	7-7
	abs-segs (Explicit Address Management) . . .	7-8
Section 8	Mechanisms.	8-1
	Policies, Protocols, and Organizations . . .	8-1
	Global Page Lock.	8-1
	Wait Events Used by Page Control.	8-4
	Wait Protocols of Page Control.	8-6
	DIM Interface and +Running+	8-11
	ALM Page Control Environment.	8-14
	Error Strategy.	8-15
	Stack Management and Interface with the Traffic Controller	8-18
	Page States	8-21
	Tracing Mechanisms.	8-29
	Individual Mechanisms.	8-29
	Waiting for the Page Table Lock	8-29
	FSDCT Paging.	8-30
	Per-Process Trace List.	8-32
	Disk Record Allocation/Deallocation . . .	8-32
	Internal Interfaces.	8-33
	Main Memory Frame Allocation.	8-35
	Replacement Algorithm Writebehind	8-35
	Page Writing/Purification	8-36
	Page Reading.	8-38
	Paging Device Record Allocator.	8-39

Page

RWS Initiator 8-40
Paging Device Housekeeping and Replacement. 8-41
Eviction Cleanup. 8-42
Per-Page Cache Management 8-43
Demand Eviction 8-43
Page abs-wiring 8-45
I/O Posting 8-46
Utility Subroutines 8-49

Section 9 Services of Page Control. 9-1
Page Fault Handling. 9-1
Services for Segment Control 9-4
Activation-Time Service 9-4
File-map/Activation Attribute Reporting . 9-5
Deactivation Service. 9-6
Call-Side PD Eviction Subroutine. 9-7
Truncation Service. 9-7
Boundsfault Service 9-8
Modified-Switch Setting 9-9
Post-Crash PD Flush. 9-10
Shutdown and Demounting Services 9-13
Record Address Depositing Services 9-14
Paging Device Record Deletion. 9-15
Forced Segment I/O and Wiring. 9-15
Abs-Wiring Service 9-17
Main Memory Deconfiguration Service. 9-18
Services for Traffic Control 9-19
Process Loading 9-19
Process Unloading 9-20
Post-Purging. 9-20

Section 10 Peripheral Services of Page Control 10-1
Procedure Wiring 10-1
Paging Device Reconfiguration. 10-2
Main Memory Frame Freeing. 10-4

Section 11 Quota Management. 11-1

Section 12 Ring Zero Volume Management 12-1
Introduction and Overview. 12-1
Concepts 12-1
Preacceptance. 12-3

Section 13 Data Bases of Ring Zero Volume Management . . . 13-1
Volume Label 13-1
Volume Map 13-5
VTOC Header. 13-6
Bad Track List 13-7
FSDCT. 13-7
Physical Volume Table (PVT). 13-10
Logical Volume Table (LVT) 13-14
PVT Hold Table 13-15

Section 14 Operations of Ring-0 Volume Management. 14-1
Acceptance of Physical Volumes 14-1
Physical Volume Demounting 14-2
Demount Protection. 14-4
Ring Zero Logical Volume Management. 14-6
Bootstrapping of Logical Volume Hierarchy (the RPV) 14-7
RPV-only Directories. 14-8
Cold Boot of the RPV. 14-8

Page

Sons-LVID Setting. 14-8
RPV-only Directory Setting 14-9
Disk_Table Location Setting. 14-9
Explicit Disk Reading, Writing, and Testing (read_disk) 14-9

Section 15 Physical Volume Salvager Interaction. 15-1
Assumptions Made Valid By the Physical Volume Salvager 15-2
Forms of Damage Corrected by the Physical Volume Salvager 15-2
Other Volume Salvager Actions. 15-3
The Disk Rebuilder 15-3
Assumptions Not Checked By the Volume Salvager. 15-3

Section 16 Scenarios 16-1
A Segment Fault. 16-1
A Page Fault, In Page Multilevel 16-4

Section 17 Glossary. 17-1

Appendix A Changes for MR 6.0. . . . . A-1
Prewithdrawing Policy. A-1
Per-process Hardcore Segment Policy. A-2
Volume Dumper Support. A-2
Page Posting Queue A-4
Page Control Traffic Control Interface . . . A-7
Page Control Consistency A-7
Page Control Error Policy. A-10
Large Volume Map Space A-12
Damaged Segments A-12
Quota Validator. A-13
Support of Hierarchy Salvager. A-16
Limited Update Backlog A-17
Partial Shutdown A-18
Other Considerations A-18

Page

ILLUSTRATIONS

Figure 5-1 The Clock Algorithm 5-10
Figure 6-1 Page Control Data Bases Page not in Main Memory or on Paging Device 6-25
Figure 6-2 Page Control Data Bases Page in Main Memory not on Paging Device 6-26
Figure 6-3 Page Control Data Bases Page in Main Memory and on Paging Device 6-27
Figure 6-4 Page Control Data Bases Page on Paging Device, not in Main Memory 6-28
Figure 6-5 Page Control Data Bases: Read-Write Sequence. . 6-29
Figure 8-1 Traffic Controller Interface Stack Management . 8-20
Figure 8-2 States of Page. 8-23
Figure 8-3 States of Page in Macro States. 8-24
Figure 8-4 Read-Evict, Write-Mod Cycles. 8-25
Figure 8-5 States of Main Memory Frames. 8-26
Figure 8-6 States of Paging Device Record. 8-27
Figure 8-7 States of Disk Address. 8-28
Figure 8-8 ALM Page Control Call Flow. 8-34
Figure 8-9 Page Control Interrupt Side, normal posting . . 8-48
Figure 8-10 Page control Interrupt Side, RWS posting. . . . 8-49
Figure A-1 Coreadd Queue Locking A-7
Figure A-2 Quota Validator A-15

SECTION 1

INTRODUCTION

This PLM describes the construction, modularization, operation, and interaction of those subsystems of the Multics supervisor that implement the Multics virtual memory. The subsystems are:

- Segment Control; responsible for maintaining the disk-resident images of segments and their attributes (the VTOC), and creating and multiplexing the Active Segment Table Entries, that allow disk-resident segments to be accessed as part of user address spaces. Segment control is responsible for performing physical operations (creation, deletion, truncation, max-length setting) upon nonactive segments, and relaying responsibility for performing these operations upon active segments.

- Page Control; responsible for bringing pages of segments in and out of main memory and the paging device (bulk store), if present. It manages the movement of all pages, and the assignment and deassignment of secondary storage addresses. Page control performs services on behalf of diverse subsystems such as traffic control (to load and unload processes at time of gain/loss of eligibility) and reconfiguration (vacating memory controllers at deconfiguration time) when use or nouse of pages of segments or frames of any kind of storage are an issue. Page control is also responsible for performing physical operations upon active segments, and implementing the main-memory sharing (page replacement algorithm of the system).

- Volume Management; responsible for the dynamic introduction and removal of physical and logical storage system volumes from the running system. It is also responsible for maintaining the integrity of volumes across multiple bootloads and crashes, and the repatriation of permanent volume-resident information in case of crash. Volume management implements as well the logical volume sharing policy, and the per-process attachment concept.

The following two subsystems, although intimately related to the storage system, are not described here.

- Directory Control; responsible for creating, maintaining, and interpreting the contents of directories, being branches for segments and directories, Access Control Lists (ACLs), names, and pointers to segment VTOC entries (VTOCEs). Directory control is accessed primarily through the user gate (hcs_) and implicitly relies upon the services of the other subsystems of the virtual memory, directories being simply segments to these subsystems.

- The directory and physical volume salvager subsystems, although not invoked during normal operation of Multics, play a critical role in ensuring the integrity of the storage system, and automatic invocation of these salvagers is relied upon to force the truth of certain predicates about disk contents. The Directory Salvager, a descendant of the old Regular Salvager of systems of earlier genre than 4.0, checks and corrects the physical structure of directory contents. The Physical Volume Salvager reconstructs critical tables on packs that must be developed from scratch after a fatal (ESD fails) crash, and ensures the consistency of VTOC entries (VTOCEs).

These subsystems are logical, rather than actual, organizations of code and data bases. Many critical and interesting programs fall into several of them simultaneously, or none exactly. These artificial functional divisions are created as an attempt to guide the description, and help the reader focus attention more precisely. Therefore, this PLM is divided into three sections, describing segment control, page control, and volume management independently.

SECTION II

SEGMENT CONTROL OVERVIEW AND CONCEPTS

Segment control is that subdivision of the Multics supervisor that is responsible for the maintenance of disk-resident segment images (VTOC entries), and the management of active segments. A large part of segment control consists of the mechanism necessary to activate and deactivate segments: another major part is the buffering and reading/writing of VTOC entries. These terms will all be clarified later.

The segment control portion of this PLM is organized in three sections:

1. Section II, Control Overview and Concepts

2. Section III, The VTOC Manager

3. Section IV, Services of Segment Control

The plan of discourse is to lead up to Section IV. Segment control, as all subsystems in a computer system, performs a set of services fulfilling a set of needs of the rest of the system. Among these services, in the case of segment control, are the activation of segments in response to segment faults, the truncation of segments, and the reporting of dynamic attributes of segments. In order to understand the implementations of the mechanisms that perform these services, detailed in Section IV, the overall organization and basic internal mechanisms of segment control must be understood. These are stated in Section IV. Included herein is a detailed breakdown of the data bases used by segment control, the ASTE, the VTOCE, and the VTOC buffer segment, and an explanation of locking policies used.

The VTOC manager is a large and important part of segment control, which is fairly well isolated. An entire chapter is devoted to its organization and implementation.

VTOC, AND DISK-RESIDENT SEGMENT IMAGES

Since release 4.0, each segment of the Multics storage system *resides* on one and only one secondary storage physical volume. This is a basic design policy that limits the amount of damage caused by the failure of one physical volume of the hardware on which it is mounted. For a segment to "reside" on a physical volume means that all of the pages of the segment are *allocated*. This means that nonzero pages of the segment are *assigned* page frames (records) on that physical volume, from which they are read, and to which they are written when and if each such page is evicted from main memory or the paging device.

Therefore, each physical volume contains a complete set of segments. This set of segments is described by the Volume Table of Contents, or VTOC of the physical volume. The VTOC is an array of fixed-length elements called VTOC Entries (VTOCEs). The VTOC is at a fixed place on each physical volume (see disk-pack.incl.pl1). Each VTOCE either describes a segment or is *free*, available for later assignment to a segment. The VTOC is of fixed size, and is created at pack initialization time.

Each segment residing on a given pack is therefore uniquely identified by the VTOC index of its VTOCE on that pack. VTOC indices are origined at zero. Therefore, the pair of physical volume and VTOC index uniquely identifies any segment in the storage system hierarchy. It is this form of identification, in the form (physical volume ID, VTOC index) that appears in directory branches. Free VTOC entries are chained in a list on each pack, the head of this list being maintained in the Physical Volume Table Entry (PVTE) while the volume is mounted or the *VTOC Header* of the pack when not. (The VTOC Header is actually a small collection of parameters such as this, kept at a fixed place on each pack. (See disk_pack.incl.pl1)).

Each VTOCE consists of three logical parts, which are designated as the *activation information*, the *file map*, and the *permanent information* of the segment. The activation information is all other information than the file map that is needed to use the segment, or more technically, to activate it. It also holds all of the information that is likely to be changed by virtue of the segment having been active (used). Such information includes some information implicit in the file map but expensive to determine, such as current length and number of records used, some information necessary for checking, such as the segment unique identifier (*UID*), and date-times of last modification and use. Quota cells and accounts for directories reside in the VTOCEs of the directories as well, among the activation information. This is because simply being active (having inferior segments gain and lose pages) can affect this information. Almost all of the activation information resides in the Active Segment Table Entry (described later) while the concerned segment is active.

The file map is an array of 256 *record addresses* or *null addresses* detailing where on the physical volume each page of the segment resides. A *null address* (not to be confused with the *nulled addresses* used internally by page control (see Section V) is an 18-bit quantity, which, when appearing in a file map, means that no record of the pack is assigned to that page of the segment, the page logically contains zeros, and does not count against quota used, or the current length of the segment. For example, when a segment is created, the file map of its VTOCE is filled entirely with null addresses as the contents of the segment is logically zero. Null addresses in VTOCE file maps are recognized by their high-order bit (400000 DU) being *ON*. The lower bits are debugging information, describing by which agency the null address was created. (See null_addresses.incl.pl1). A *record address* is the address of a record of the physical volume. All volumes are divided into key-word records, and start at record zero. It is one of the design goals of page control that no record address *ever* appears or is allowed to remain in a VTOCE file map unless it is known for a fact that data from that page actually appears on the physical pack; this eliminates the possibility of windows during which if the system crashed, the VTOCE file map would describe a record containing uninitialized data, potentially a security problem.

The permanent information in a VTOCE consists of attributes that are either determined forever at segment creation time, or rarely changed. Such information includes the unique ID pathname (array of segment unique IDs of superior directories) access class, date/time dumped by the physical volume dumper, and the primary segment name, placed there only for debugging and the physical volume salvager.

The structure of a VTOC entry in detail is spelled out below. The current VTOC entry is 192 words long, consisting of three sectors of MSU0400 or MSU0451 disk. Most of this entry is the file map (128 words). Thus, most accessing of VTOCEs deals only with the activation information and a small portion of the file map (most segments are only a few records long). Therefore, VTOCEs were organized such that the activation information (about 20s10S words) is at the beginning of the VTOCE, followed by the file map, and then the permanent information. This makes it so that most interactions with VTOCEs deal with only the first few (say 30s10S) words. In order to take advantage of this fact, VTOCEs are accessed via sector-by-sector I/O, as opposed to residing in pages of segments. Were the latter the case, each reference to a VTOCE would require paging in 1024 words when perhaps as few as thirty, or at most 192, were needed. A large complex mechanism (the VTOC Manager, vtoc_man) and program exist to manage these sector-by-sector I/Os and their buffering. However, the physical volume salvager and other subsystems, notably BOS SAVE, prefer to deal uniformly with pages. In the case of the physical volume salvager, this allows it to use read-ahead entries in page control to optimize performance. Therefore, the VTOC is laid out in pages, such that any VTOCE can be accessed by reading/writing a given record, preferably by accessing it via paging, so as to leave the other VTOCEs unaffected. This allows five and one-third VTOC entries per page (1024/192). Due to the possibility of having pages split across cylinders, which would create "slow" pages, Multics does not use fractional pages at ends of cylinders. Therefore, if VTOCEs were packed 5-1/3 per page, some VTOCEs would not in fact be contiguous on the disk, eliminating the possibility (not now realized) of single-operation I/O in a uniform manner to transfer an entire VTOCE. Thus, VTOCEs are packed five per page, with a 64-word unused region at the end of each page. Each VTOCE therefore consists of three (192/64) contiguous 64 word sectors. These sectors define three physical regions of the VTOCE, or vtoce-parts; known as Part I, Part II, and Part III. Part I contains the activation information and the start of the file map, Part II the middle of the file map, and Part III the end of the file map and the permanent information. Thus, most VTOCE transactions consist of reading or writing Part I, 64 words, 1 sector, of some VTOCE.

We now consider the individual items in a VTOC entry (VTOCE), with some discussion of their significance.

```
dcl 1 vtoce based (vtocep) aligned,


    (2 next_free_vtocx fixed bin (17),
     2 incr_dmpr_thrd fixed bin (17),

     2 uid bit (36),

     2 msl bit (9),
     2 csl bit (9),
     2 records bit (9),
     2 pad2 bit (9),

     2 dtu bit (36),

     2 dtm bit (36),

     2 nqsw bit (1),
     2 deciduous bit (1),
     2 nid bit (1),
     2 dnzp bit (1),
     2 gtpd bit (1),
     2 per_process bit (1),
     2 pad3 bit (12),
     2 dirsw bit (1),
    2 master_dir bit (1),
    2 pad4 bit (16),

    2 infqcnt (0:1) fixed bin (17),

    2 quota (0:1) fixed bin (17),

    2 used (0:1) fixed bin (17),

    2 received (0:1) fixed bin (17),

    2 trp (0:1) fixed bin (71),

    2 trp_time (0:1) bit (36),




    2 fm (0:255) bit (18),

    2 pad6 (10) bit (36),

    2 ncd bit (1),
    2 pad7 bit (17),
    2 cons_dmpr_thrd fixed bin (17),

    2 dtd bit (36),

    2 volid (3) bit (36),

    2 master_dir_uid bit (36),




    2 uid_path (0:15) bit (36),

    2 primary_name char (32),

    2 time_created bit (36),

    2 par_pvid bit (36),

    2 par_vtocx fixed bin (17),
    2 branch_rp bit (18)) unaligned,

    2 cn_salv_time bit (36),

    2 access_class bit (72),
    2 checksum bit (36),
    2 owner bit (36);
```

Activation Information

next_free_vtoce
is meaningful only in free VTOCEs. It is the VTOC index of the next free VTOCE in the free VTOCE chain. Note that -1 is the end of the chain. In an occupied VTOCE, this field is zero.

incr_dmpr_thread
is not used.

uid

is the segment unique identifier, assigned at segment creation time. This matches an identical field in the directory branch for the segment. It must be zero in a free VTOCE, and zero UID implies a free VTOCE. This quantity is checked every time the VTOCE is used, to check that the right VTOCE is being accessed, and that no damage has occurred to the VTOC or the pack. Failure of the segment unique ID (UID) to check is known as a connection failure.

msl

is the maximum segment length, in pages. This information is put into the SDW (segment descriptor word) of a process handling a segment fault.

csl

is the current length of the segment, in pages. This may be defined as one plus the index (starting at zero) of the highest nonnull address in the file map. The physical volume salvager computes it this way. The most interesting property of vtoce.csl is that it tells those reading the VTOCE whether or not they have to read Part II, or even Part III, to acquire the entire nonnull portion of the file map.

records

is the number of nonnull addresses in the file map. Again, this is computed by evaluating this criterion by the physical volume salvager. This number may also be viewed as the number of quota units consumed by the segment. When the segment is active, a parallel quantity is maintained by page control, and periodically updated to vtoce.records. Since there can be records that count against quota that do not appear in the VTOCE file map yet, as they have not been written, (see the discussion of record address above), the statement "Records used changed from <number> to <smaller number>" by the VTOC salvager indicates that a segment has lost pages in this way. This number exists to avoid the necessity to recompute it every time the segment is activated, as page control needs it.

dtu

is the "file system time" (upper 36 bits of real-time clock) recording the "date-time used" attribute of the segment. Other than segments activated with "transparent usage" (such as by the Hierarchy Dumper), this is generally the time that the VTOCE was last updated (from the AST).

dtm

is the file-system time recording the "date-time modified" attribute of the segment. This quantity is maintained by page control (as aste.dtm) when the segment is active. It, like other activation attributes, is updated from the Active Segment Table.

nqsw

is a switch indicating that page control should suppress checking of quota overflow for this segment. This switch is never intentionally turned on in a VTOCE; it is simply a reflection of an AST switch used for certain special segments.

deciduous

similarly is a reflection of an AST switch, which is never, and cannot be explicitly turned on in a VTOCE. It marks the VTOCE of a deciduous segment, primarily so that the physical volume salvager may reclaim pages of such segments. A full discussion of deciduous segments is given in the Multics Initialization PLM, Order No. AN70. The definition is repeated here:

> A deciduous segment is one that is loaded by system initialization in collections 1 or 2, is part of the global or initializer's hardcore address space, and acquires a branch in the hierarchy, via the program init_branches in collection 2.

nid

for "no incremental dump". A so-called "VTOC Attribute" (see later discussion of "VTOC Attributes"), restraining the physical volume dumper from dumping this segment in an incremental dump.

dnzp

for "don't null zero page". Both a "VTOC Attribute" and used for deciduous and other special-case segments. When this segment is active, the AST reflection of this bit (aste.dnzp) prevents page control from detecting, and thus scheduling for deposit, pages of zeros. A zero page of a "dnzp segment" is as good as any other page. This is necessary for "PTW-level abs-segs" and the prewithdrawing policy (see Section VII).

gtpd

for "global transparent to paging device". Prevents pages of this segment from migrating to the paging device (bulk store subsystem). Just about everything said for vtoce.dnzp is true for vtoce.gtpd as well.

per_process

developed at VTOCE creation time and at update time. If on, the segment owning this VTOCE is either >process_dir_dir or a descendant of a segment with vtoce.per_process on. Principal use of this bit is to allow the physical volume salvager to discard such VTOCEs and free the pages they claim.

dirsw

identifies the VTOCE of a directory. Principally informative, it must check with the directory switch in the branch of the segment at activation time, or a connection failure is indicated. Biases the physical volume salvager in favor of this segment in resolving page conflicts.

master_dir

marks the VTOCE of a master directory. This is necessary to facilitate the redistribution of quota at directory deletion time: the delete_vtoce program must know whether or not to pass quota back up based on this bit. (See "Segment Deletion".)

infqcnt

previously count of inferior directories with quota accounts, for a directory VTOCE, this field is now considered obsolete.

quota

is the amount of quota assigned to the directory (must be the case if nonzero) owning this VTOCE. Like vtoce.infqct, vtoce.used, vtoce.received, vtoce.trp, and vtoce.trp_time, this field is actually a two-element array, the zeroth (left-hand) element for segment quota, and the first, (right-hand) for directory page quota, currently partially implemented.

used

is the amount of quota used by inferior segments and directories, (see vtoce.quota above). It can be recomputed only by recursively summing the vtoce.records fields of all VTOCEs for segments inferior in the hierarchy. This is the number reported by hcs_$quota_get (the get_quota command, for example) as used, it does not include used totals of inferior accounts. Maintained for active segments by page control, vtoce.used is derived from the ASTE. Validly nonzero only for directory VTOCEs.

received

is the sum of the quota given to this (directory) and the vtoce.received for all inferior directories, if any. Of course, validly nonzero only for directory VTOCEs. This quantity is necessary in order to determine if any quota has been delegated below any point of the hierarchy. It is a peculiar quantity (also true of vtoce.trp) in that it is one of two items in the VTOCE activation information that must be read in from the VTOCE, i.e., cannot be derived solely from bits and fields of the Active Segment Table, at VTOCE update time. This field, like vtoce.trp and vtoc.trp_time, is only used for directories with quota accounts, i.e., vtoce.quota (0 or 1)≠ 0.

trp

is the page-second time-record usage product for the quota-account-owning directory that must own this VTOCE. See vtoce.received, above.

trp_time

is the file-system time at which vtoce.trp was updated; this is always the time of a VTOCE update (see "VTOCE Updating," in Section IV).

File Map

fm

is the array of packed, 18-bit null addresses and record addresses describing which pages of the segment owning this VTOCE are logically nonzero, and where the images reside. The interesting (containing other than null addresses) extent of the file map is told by vtoce.csl. Those who need the file map are satisfied not to read the particular null addresses that may appear; the differences between the types of null addresses is solely for debugging.

Permanent Information

ncd

for "no complete dump". Treated like a "VTOC Attribute". When on, restrains the physical volume dumper, when performing a complete dump, from dumping the segment owning the VTOCE. Among the permanent information (in Part III) due to the relative infrequency of complete dumps.

cons_dmpr_thrd

is not used.

dtd

is the file-system time that this VTOCE, and its segment, were dumped by the physical volume dumper.

volid

is an array of backup medium identifiers, set by the physical volume dumper, identifying the volumes of backup medium (tape) on which the last incremental, consolidated, and complete dumps of this segment and its VTOCE were performed. Inspection of those volumes produces maps giving earlier volumes, and so forth through the life of the segment.

master_dir_uid

is the segment UID of the master directory against whose master directory quota account the pages of the segment owning this VTOCE are counted. This information is used by master directory control, and is updated by the hierarchy salvager, if necessary, when running in connection-checking mode.

uid_path

is an array of the Segment Unique IDs (UIDs) of all *directories superior to this segment*. Thus, the zeroth element of vtoce.uid_path for every VTOCE in the system except the VTOCE of the root (>) is the *UID of the root* ("777777777777"b3). The VTOCE of a son of the root (e.g., >user_dir_dir) contains only one element, the UID of the root, etc. The UID of the segment owning the VTOCE, which appears among the activation information in Part I, is *not* in vtoce.uid_path. This UID path places the VTOCE exactly in the hierarchy. It is only used explicitly by master directory control, to identify directories that have been given master directory quota accounts, in a manner insensitive to renaming of these directories. It is checked and corrected (given that *forward connection failure*, the kind described previously, does not exist), by the hierarchy salvager when running in VTOCE-checking mode. The array vtoce.uid_path can also be used, if assumed accurate, to determine if a segment has no branch, no parent, or no grandparent, etc. Such a segment, which can arise in certain crash situations and salvaging situations, is called an *orphan*, and is said to suffer a *reverse connection failure*. The online pack utility sweep_pv is capable of locating and deleting such VTOCES, which can tie up pages. (See "Special Services for sweep_pv" in Section IV.)

primary_name

is the name appearing in the branch for the segment at the time the segment was created. Ordinary rename operations will not update vtoce.primary name, due to the expense of reading and writing Part III to update permanent information. The hierarchy salvager, running in VTOCE-checking mode, however, will. The name in the VTOCE is never seen by users. The physical volume salvager prints it out when VTOCE problems are encountered. Since it is not accurate, it is only a clue to the identity of the segment. As long as the VTOCE was not freed by the physical volume salvager, the vtoc_pathname tool may be given the volume name and VTOC index printed out by the physical volume salvager. The BOS SST name table filler (SSTN) also picks up these names and puts them in the segment sst_names_ at crash time. Thus, it is these names that appear in BOS dumps and FDUMPS.

time_created

is the file-system time at which the VTOCE (and therefore the segment owning it) was created. Principally of historical value (sweep_pv reports it when deleting orphans).

par_pvid

is the physical volume ID of the volume containing the *directory* containing the segment owning this VTOCE. Not transparent to segment-moving (see "Segment Moving" below), this field is set, but not now used.

par_vtocx

is the VTOC index of the VTOCE of the *directory* containing the segment owning *this* VTOCE in its physical volume. As vtoce.par_pvid above, it is not transparent to segment moving and not currently used.

branch_rp

is the relative offset of the directory branch describing this VTOCE in its directory. Intended for debugging, it is maintained by the hierarchy salvager operating in VTOCE-checking mode. Note that online salvaging of a directory causes branches to move around.

cn_salv_time

is not currently used. It was intended to be the time at which lack of reverse-connection-failure was last checked by the reverse-going (branch-checking) mode of the physical volume salvager, since decommissioned.

access_class

is the AIM access class of the segment owning this VTOCE.

checksum

currently not used.

owner

intended to be the physical volume ID of the volume on which this segment and its VTOCE reside, this field is not used.

ACTIVE AND NONACTIVE SEGMENTS

The VTOC entry and the records designated by its file map are the permanent record of a segment on disk. They are the entire and accurate record of the segment when the pack is not mounted or the system is shut down. In order for a segment to be accessed via the hardware, it must have a page table in main memory, and much of the VTOC information, specifically the file map and activation information, must be in main memory where page control can use it to resolve page faults, and modify it as pages are created and zeroed. A segment in this state is called an active segment. A segment not in this state is called a nonactive segment. The repository of activation information for a segment is the system data base, the Active Segment Table (AST). This table, which resides in the System Segment Table (SST), consists of AST entries (ASTEs). An ASTE contains, when in use, the activation information for one segment. Following each ASTE, part of the ASTE in some sense, although not part of the ASTE proper, is the page table for that segment. The page table is maintained by page control, which uses and updates the activation information resident in the ASTE as the segment is used. The file map is handed to page control by placing it in the page table.

The AST is an unpaged data base. Since it is finite, the number of AST entries is limited. Currently, there are four fixed sizes, those whose page tables can describe 4, 16, 64, and 256 pages respectively. The AST is thus divided into four pools, whose sizes are set by the four specifications on the SST CONFIG card, a critical system tuning parameter. Since we have just defined activity as the state of having page table and activation information in main memory, and this is a precondition for use of the segment, only active segments can actually be addressed by the hardware. Thus, all segments must be made active before they can actually be used. Therefore, the fixed number of AST entries must be multiplexed among all of the segments in the hierarchy. It is one of the prime responsibilities of segment control to multiplex this resource. When an attempt is made to reference a segment that is not active (this is one of the possible outcomes of a segment fault), the segment must be activated, or made active (given an ASTE, and the activation information and file map copied out of the VTOCE into it). If there are no free ASTEs of the appropriate size available, some segment must be deactivated to free an ASTE. This deactivation consists of making the segment inaccessible to user processes, evicting all pages of the segment from main memory and the paging device, updating the VTOCE by copying the (possibly modified) activation information back into it from the ASTE, depositing nulled addresses (see "Address Management Policy", Section VII), and freeing the ASTE. Once this has been done, the segment deactivated is in the same state as one that has not been activated, and a segment fault and subsequent activation result from an attempt to reference it. Choosing a proper segment to deactivate is a complex issue that must choose that segment which will probabilistically and heuristically be reactivated at the furthest time in the future. The algorithm used to make this choice (in the program get_aste) is described further on under "AST Replacement Algorithm" in this section.

There are segments that are active during the entire life of a bootload; all hardcore supervisor and all deciduous segments are this way. These segments are used by software, such as the virtual memory control software being described here, that are not dependent upon the dynamic activation/deactivation features that they implement in order to operate; similarly, the page control software does not itself take page faults. There are segments that may not be deactivated for long periods of time: such segments are the PDS (Process Data Segment) and KST (Known Segment Table) of processes, for they become part of the supervisor in some processes, and thus are used to implement the virtual memory in that process. There are segments, namely the paged, nondeciduous segments of the supervisor, and the descriptor segments of processes, that do not have VTOCEs, but only have ASTEs. They are always active.

VTOC ATTRIBUTES

When a normal, VTOCE-owning segment is nonactive, the VTOCE is the repository of the file map and activation information. All requests for this data must go to the VTOCE of the segment. When a segment is active, however, the ASTE is the only valid repository of this information. Information such as current segment length can change as processes store data into the segment. Quota used can change as such operations are performed on segments inferior to a given directory.

User-interface programs, and directory control, who have need to know activation attributes must therefore go to either one of two places to get these attributes. In order to localize this knowledge, all programs outside of segment control that need to ascertain or set activation attributes of segments call the procedure vtoc_attributes at one of its many entry points to obtain or set this information. This procedure determines whether or not the segment is active (see "AST Hash Table and Determining Activity" below), and inspects or modifies the appropriate data object. These attributes, which have been called "activation attributes" in the context of the VTOCE, are called "VTOC attributes" in the context of other storage-system features such as bit count, access mode, etc. It is through this means, for instance, that hcs_$status_long (through the hardcore module "status") obtains current length/records used for segments.

AST HASH TABLE AND DETERMINING ACTIVITY

Every segment that has a branch in the hierarchy (this excludes nondeciduous hardcore segments, unpaged supervisor segments, descriptor segments, and PRDSs) can either be active at any instant or not. A process that attempts to use such a segment, by performing a segment fault upon it, must determine whether or not it is active. If it is, it is a simple matter to add an SDW (Segment Descriptor Word) describing the page table in the segment's ASTE to the descriptor segment of that process. If not, the segment must be activated (which may, as outlined above, entail deactivating other segments) before an SDW can be so added. Similarly, vtoc_attributes must know whether or not a segment is active to know where to obtain or change these parameters. Thus, a hash table is kept, called the AST Hash Table, which locates the ASTE of any active segment, or the fact that it is not active. This table is an array of thread heads, kept in the internal static of the procedure search_ast (in the supervisor, this makes it a global data base as opposed to per-process internal static) (but also locatable from the pointer sst.asthp for debugging and dump analysis). Each bucket starts a list (which ends in zero) of AST entries the UIDs of whose segments have the same low six bits. Thus, given the UID of any segment, we can find the bucket numbered by the low six bits of this UID, and chase the thread (through the field aste.ht_fp) until either a zero is encountered (segment not active), or an ASTE whose field aste.uid contains the UID we have been given, in which case this is the ASTE for that segment, and of course, it is active.

The AST hash table is protected by the AST lock (see "Segment Control Locking Policies" below). Deciduous segments are hashed into this table as soon as they acquire branches, at which point they acquire the UID in that branch and stay hashed in for the life of that bootload.

AST HIERARCHY

The root directory (>) cannot be deactivated. Other than that, no segment may be active unless its parent is active. This is so because the quota account parameters against which a segment's records-used are charged is maintained in (is an activation attribute of) the ASTE of one of its ancestors (its parent, or that one's parent, etc.). Another reason for requiring the activity of parents is that date-time modified for directories is in fact date-time modified for the last-modified segment in the subtree rooted at that directory; this allows the hierarchy dumper to determine if a subtree need be walked by inspecting the date-time modified of its root. Keeping date-time modified, a VTOC (activation) attribute up to date for a straight line back to the root, requires all directories in that line to be active, so that page control can modify this attribute. Thus, it is necessary that each ASTE have a pointer to its parent's ASTE (the root has zero in this field, otherwise like all pointers in the SST segment other than aste.strp, it is a relative offset into the SST segment). There exists an operation called a boundsfault, wherein a segment grows, and requires a larger ASTE. Should this happen to a directory with active inferior segments and directories, all of the parent-pointers in the inferior ASTEs would become wrong when the directory changed ASTEs. Therefore, a first-son-brother thread is maintained among ASTEs, so that all inferior ASTEs can be located in the case of a boundsfault. This technique is also used at segment-move time (see "Segment Moving", below).

BREAKDOWN OF THE AST ENTRY

The following is a detailed discussion of all of the fields and bits in an ASTE (AST entry). Remember that many of these fields and bits are but reflections of similar fields in the VTOCE. Such fields are marked with an (*).

```
dcl 1 aste based (astep) aligned,

    (2 fp bit (18),
    2 bp bit (18),

    2 infl bit (18),
    2 infp bit (18),

    2 strp bit (18),
    2 par_astep bit (18),

    2 uid bit (36),

    2 msl bit (9),
    2 pvtx fixed bin (8),
    2 vtocx fixed bin (17),

    2 usedf bit (1),
    2 init bit (1),
    2 gtus bit (1),
    2 gtms bit (1),
    2 hc bit (1),
    2 hc_sdw bit (1),
    2 any_access_on bit (1),
    2 write_access_on bit (1),
    2 inhibit_cache bit (1),
    2 explicit_deact_ok bit (1),
2 pad1 bit (9),
2 ehs bit (1),
2 nqsw bit (1),
2 dirsw bit (1),
2 master_dir bit (1),
2 pad4 bit (1),
2 tqsw (0:1) bit (1),
2 ic bit (10),

2 dtu bit (36),

2 dtm bit (36),


2 quota (0:1) fixed bin (17),

2 used (0:1) fixed bin (17),

2 csl bit (9),
2 fmchanged bit (1),
2 fms bit (1),
2 npfs bit (1),
2 gtpd bit (1),
2 dnzp bit (1),
2 per_process bit (1),
2 pad2 bit (3),
2 records bit (9),
2 np bit (9),


2 ht_fp bit (18),
2 fmchanged1 bit (1),
2 pcos bit (1),
2 pack_ovfl bit (1),
2 pad3 bit (7),
2 ptsi bit (2),
2 marker bit (6)) unaligned;
```

aste.fp

is the forward pointer (rel pointer in SST segment) to the next ASTE in the so-called "used list". There is one used list (ASTE chain) for each pool (size) of ASTE. Free ASTEs are at the head of this chain, others follow. Some nondeactivatable ASTEs are not in the list, such as supervisor segments (including deciduous ones), descriptor segments, and PRDSs. There are special lists for special segments. See "AST Replacement Algorithm".

aste.bp

is the backward pointer to the previous ASTE in the appropriate used list.

aste.infl

for "inferior list", is a (relative) pointer to the next ASTE in a list of ASTEs whose segments have the same parent as the ASTE of this segment. We will contract this terminology to say "a list of ASTEs who have the same parent ASTE". See "AST Hierarchy" above. This is really a "brother's list".

aste.infp

is a (rel) pointer to the first ASTE in the list (through aste.infl, described above) of ASTEs of which this ASTE is the parent. Like all ASTE lists and pointers, it is zero if there is none.

aste.strp

is a relative pointer to the first *trailer* in the *system trailer segment*, str_seg, zero if there are none, for this ASTE. An ASTE acquires a trailer for each SDW constructed via a segment fault, which describes the page table in this ASTE. It facilitates revocation of SDWs when the segment is deactivated, deleted, or suffers an access change (see "Trailers and Setfaults" below). For nondeciduous supervisor and initialization segments, this system-wide segment number is stored here.

aste.par_astep

is a relative pointer to the *parent* ASTE of *this* ASTE, if this ASTE is for any segment in the hierarchy other than the root directory (>). Page control uses this quantity to chase up the hierarchy to find quota cells at page creation time, and to update aste.fms (see below) up the hierarchy to trigger the hierarchy dumper.

aste.uid

*is the UID of the segment owning this ASTE. It agrees with vtoce.uid, which must be the same as the UID in the directory branch. Not only is this field necessary to allow the AST hash table to be used, but is necessary to reconstruct Part I of the VTOCE at deactivation/update time without reading it, as the UID of the segment is among this information.

aste.msl

*is the maximum segment length in pages. An activation attribute, attempted *connections* to this segment at segment fault time check their address of reference against this quantity, and, shifted appropriately, it is placed into the SDW constructed. (See "Segment Fault Handling".)

aste.pvtx

is the Physical Volume Table Index (PVTX) for the mounted physical volume on which this segment appears. See the discussion of the Physical Volume Table in Section XIII. This number identifies a mounted physical volume.

aste.vtocx

is the VTOC index of the VTOCE of the segment owning this ASTE on the physical volume on which it resides. This is gotten from the directory branch for the segment, and is used to specify the VTOCE of the segment at deactivation/update time.

aste.usedf

when on, differentiates an in-use AST entry from a free one. See "AST Replacement Algorithm" below.

aste.init

turned on by page control when the last page of a segment migrates out of main memory. One of the inputs of the AST replacement algorithm. Turned off when any page comes in. (See "AST Replacement Algorithm" for motivation.)

aste.gtus

*(A VTOC attribute) "global transparent usage switch". When this is on, the segment is in "transparent usage". This means that the date-time used in the VTOC entry is saved in aste.dtu and put back intact at deactivation time, thus leaving no evidence that the segment had been used. The hierarchy dumper causes all segments it dumps to be activated for "transparent usage" by setting switches in its KST. This allows the dumper to run without advancing the date-time used of segments it dumps. Like aste.gtms and aste.dnzp below, this segment attribute is cumulated as processes connect (to satisfy segment faults on, construct SDWs for) this segment.

aste.gtms

*see aste.gtus above. "global transparent modified switch" causes page control not to set the file modified switch", thus preventing advancing of aste.dtm (date-time modified) as modification of pages is noticed. This is used principally for directories, whose date-time modified is not the time that they were stored into, but the time that either directory control deems that they were modified (calls sum$dirmod) or inferior segments were modified.

aste.hc

is set for ASTEs of segments created by initialization (supervisor and initialization segments) that are neither deciduous nor unpaged. These are unthreaded and delete-at-shutdown segments. See the Multics Initialization PLM, Order No. AN70. This bit is principally historical.

aste.hc_sdw

is on for all ASTEs for segments created by initialization, deciduous, delete-at-shutdown, or unthreaded. If aste.uid (and therefore segment is in the hierarchy), this segment is deciduous. Therefore, this bit reflects into the VTOCE as vtoce.deciduous.

aste.any_access_on
aste.write_access_on

are the encacheability control bits. The following table describes the number and access of all SDWs pointing at this segment (used only for segments for whom SDWs are created by segment faults):

aao	wao	
0	0	No SDWs point at this segment.
1	0	One or more SDWs describe this segment. None of them allow write access.
1	1	Exactly one SDW describes this segment. It allows write access.
0	1	More than one SDW describes this segment. At least one of them allows write access.

See "Encacheability Control" later in this section.

aste.inhibit_cache

prohibits the resetting of the encacheability bits to state "00" above upon "set acl" or "set max length" operations (setfaults). Used for I/O buffer segments that are not encacheable because of IOM access, not multiprocessor sharing. See "Encacheability Control" and "Trailers and Setfaults" below.

aste.ehs

is the entry-hold switch. Although many entries that may not be deactivated are threaded out of the AST used lists, some segments acquire and lose this property dynamically, such as PDSs and I/O buffer segments. This bit is placed on for all segments in the used lists that may not be deactivated, and causes the AST replacement algorithm to skip this ASTE. It is also put on in all segments that have aste.hc_sdw (see above) for consistency. It also has an effect upon the interpretation of aste.dnzp (see below).

aste.nqsw

*suppresses quota checking on this segment. On for all segments that have no parent, such as supervisor segments, all initialization and initialization-created segments, and the root. Notably, this flag prevents page control from chasing a nonexistent parent pointer at page creation time.

aste.dirsw

*on for a directory's ASTE. Used for metering, and at deactivation/VTOCE update time to make decisions about quota parameter updating.

aste.master_dir

*Same as vtoce.master dir, which see.

aste.tqsw

an array, one for each kind of quota. Says that this is the ASTE of a directory with a terminal quota account. Causes page control to stop looking upward and check here when making a record-quota overflow decision. Tells VTOCE updater to read in Part I in order to get time-record product parameters in order to update them.

aste.ic

is the count of inferior ASTE entries. This nonzero parameter is an input to the AST replacement algorithm (simply if nonzero). Since aste.infp has the same information, this field is superfluous.

aste.dtu

*is the file-system date-time used copied from the VTOCE field of the same name. Normally, vtoce.dtu is set to the time of VTOCE update; it is only for segments activated in "transparent usage" (see aste.gtus above) that this field is updated, unchanged, to the VTOCE.

aste.dtm

is the file-system date-time-modified, initialized by reading in vtoce.dtm at activation time. This field is advanced to the current time every time aste.fms (see below) is seen on. This includes all VTOCE updates, and whenever vtoc_attributes asks for this value. The advanced value is set back in the VTOCE at deactivation/update time.

aste.quota

*is an array (segment quota, directory quota) with the same meaning as vtoce.quota, the quota account values of a directory that has one.

aste.used

*is similarly the reflection of vtoce.used. When aste.used tries to surpass aste.quota, and aste.tqsw is on (all for segment or directory quota consistently), a record quota overflow will occur. The aste.used field, as vtoce.used, has totals for all segments (or directories) below this point for <u>any</u> directory, not only those with quota accounts.

aste.csl

*is the current length of the segment, in pages. It is maintained by page control as the end of the segment goes up and down.

aste.fmchanged

is the "file map changed" bit. This bit is put on by page control any time the state of the file map of the segment has been changed. This happens at page allocation time and page address resurrection time, as well as at zero detection time. The fact that address reporting to the VTOCE is inhibited (see "Address Management Policy" in Section PC) causes the creation of a page to trigger a VTOCE update aste.npfs

the "no page fault switch" causes page control not to honor page faults on this segment, but convert them into segment faults. It is never set except gratuitously, and is obsolete.

aste.gtpd

*"Global transparent to paging device" causes page control not to allow pages of this segment on the paging device. Its principal uses are for abs-segs, where paging is being used to address portions of disk as opposed to implementing segments, and as a user-settable performance control (as a VTOC attribute).

aste.dnzp

*"Don't null zero page". Causes page control not to recognize zero pages. See the remarks under vtoce.dnzp When aste.dnzp and aste.ehs are on cojointly, this bit causes pc$get_file_map, which reports file maps and activation attributes to update_vtoce, to not notice nulled addresses, but to leave them in the page table. This prevents the trickle update (see "AST Trickle" below) from negating the effects of prewithdrawing PDSs (Process Data Segments) (see "Address Management Policy" in Section VII).

aste.per_process

*is used to get vtoce.per_process, and for metering. It also propagates recursively.

aste.nid

*for "no incremental dump". Same as VTOCE bit vtoce.nid. Tells the volume dumper, when running an incremental dump, that incremental backup of this segment is not to be performed.

aste.ncd

*for "no complete dump". Same as VTOCE bit vtoce.ncd. Tells the volume dumper, when running a complete dump, that complete dumping of this segment is not to be performed.

aste.explicit_deact_ok

Constructed from KSTE bits of all processes connected to this segment, this bit allows the procedure demand_deactivate to explicitly deactivate the segment in response to a user call to phcs_$deactivate, generally on behalf of the hierarchy dumper. Only if all processes connecting to this segment have this bit on in the KST does it remain on in the ASTE.

aste.records

*is the number of records (pages) used by this segment. Typically, this quantity is loaded from VTOCE quantity. The only reason for this quantity is its use as a user-readable VTOC attribute, available without scanning the page table.

aste.np

Number of pages in main memory. Used solely as an input to the AST replacement algorithm. Maintained by page control. The aste.init field is turned on when this becomes zero.

aste.ht_fp

forward pointer in the AST hash chain of ASTEs with UIDs of the same low six bits. Zero at end of chain. See "AST Hash Table and Determining Activity" above.

aste.fmchanged1

this bit is turned on when aste.fmchanged is turned off, and turned off by update_vtoce when the VTOCE has been updated. Should the system crash between the turning off of aste.fmchanged and the turning off of aste.fmchanged1, the presence of the latter will signify to emergency-shutdown to reinstate the bit aste.fmchanged, for in fact, this critical bit has been turned on and the VTOCE possibly not updated.

aste.pcos

page control out-of-service. Not used yet, this bit causes a segment fault error with code error_table_$seg_busted when an attempt is made to connect to this ASTE. This will be used to notify users when the system has committed an error upon the segment.

aste.pack_ovfl

is turned on by page control when an attempt to allocate a new page for this segment has failed. In this case, page control faults the SDW for the segment, and restarts the fault. This causes a segment fault to occur, and the segment fault handler, noticing aste.pack_ovfl, invokes the segment mover to initiate a segment move. (See the general discussion, "Segment Moving" below.)

aste.ptsi

is the page table size index, 0, 1, 2, or 3, being the index of the AST pool to which this ASTE belongs. This and aste.marker, below, are attributes of the ASTE even when empty.

aste.marker

always contains "02"b3, which can never be the last six bits of a PTW (page table word). This used to be used for searching backwards through PTWs for the end of the ASTE, but has not since ASTE pointers began to appear in the core map. It is now looked at by the AST walking loop of demount_pv, simply as a check that it has not gone awry due to destroyed parameters in the SST header.

AST LISTS AND THREADS

AST entries may be threaded onto one of several lists, via the relative pointers aste.fp and aste.bp, or none at all. There are seven such lists; auxiliary lists such as the hash threads and father-son-brother lists are not under consideration in this discussion. These lists are the four "used" lists, the "init" seg list, the "temp" seg list, and the "hardcore" list. The four "used" lists, as mentioned above, contain all free ASTEs and those managed by the AST replacement algorithm. The "init" and "temp" seg lists receive "init" and "temp" segs of initialization (See Multics Initialization PLM, Order No. AN70), allocated and placed there by the initialization ASTE allocator, make_sdw. These lists are traversed at the end of initialization and the end of each collection of initialization in order to delete these segments, deletion in this case being tantamount to freeing of the ASTEs and the records allocated to these segments.

The "hardcore" list, which used to contain all nondeciduous segments loaded by initialization that were not "init" or "temp" segments, now contains only those that are deleted at shutdown time, for only these need be sought out. These "delete-at-shutdown" segments are large segments that obtain record allocations as parasites on the Root Physical Volume (RPV) instead of being prewithdrawn against the hardcore partition. Thus, in a successful shutdown situation, their records must be relinquished. See "Address Management Policy" in Section VII for full details of this mechanism.

The four AST "used" lists thread all free and replaceable ASTEs of each (pool) size. The array of four rel-pointers in aste.level.ausedp points to either the first free ASTE in the list, if any, or the first candidate for inspection for replacement if there are none. All of the free ASTEs are contiguous in the list. All of the AST lists are double-threaded circular lists: therefore, in the used lists, aste.bp of the ASTE pointed to by aste.level.ausedp of this pool is the one that is the last candidate for inspection by replacement.

It is useful to note that all active segments in the hierarchy are in the four used lists, except the deciduous segments, for it is known at the time deciduous segments are created that they will never be deactivated or subject to deactivation. The deciduous segments, therefore, have their ASTEs threaded out.

AST REPLACEMENT ALGORITHM

The AST replacement algorithm is that algorithm, implemented in the procedure get_aste, that returns a free ASTE in a given pool on demand. When there are no free ASTEs in the appropriate pool, this algorithm must select an active segment for deactivation. Since activating segments is expensive, it is advantageous to this algorithm to choose those segments to deactivate that will cause the fewest number of reactivations per time. This is a classic example of a demand replacement multiplexing algorithm, identical in purpose to page replacement algorithms, and index register management algorithms in compiler code generators, and the area is well covered in the literature. It can be shown that the best choice of segment to deactivate is the one that will next be used furthest in the future; this result follows from classic work in this area.

Of course, it is impossible to predict, in a general-purpose computer utility, the future use patterns. Therefore, the replacement algorithms try to predict the future based on the past. The AST replacement algorithm under consideration uses list position in the used list and number of pages in main memory as indications of frequency and intensity of use; the more lightly and less recently used, the lesser the indicated probability that the segment will be needed in the near future. Number of pages in main memory is also an important factor to consider in choosing a candidate for deactivation because work (page writing) is required for the modified fraction of such pages, to evict them from main memory.

The following is a description of the AST replacement algorithm. For full details, read the listing of get_aste.

If there are free ASTEs of the needed size available, return the first one, moving aste.level.ausedp at the appropriate level forward one, to make the next (possibly free) ASTE available to the next invocation of the algorithm. This also puts the returned ASTE in the least likely position for replacement, should the caller of get_aste decide to leave it there. This is consistent with the fact that the segment that will own the ASTE is now being used.

If there are no free ASTEs available, the used list at the required pool level is circumnavigated possibly several times: essentially once to find a segment with 0 pages in main memory, that failing, then for a segment with 1 page in main memory, then 2, etc., etc., until a number equal to the page table size of the pool is reached. In each pass, segments with fewer than the sought number of pages in main memory (not seen earlier because the system is moving while all this goes on) are accepted, too. When such a segment is found, it is thus, modulo the window mentioned above, one of the segments with the fewest number of pages in main memory, in that used list. This segment is chosen for deactivation, and deactivated via a call to the procedure "deactivate". The newly-freed ASTE (deactivation frees the ASTE) is returned.

When the list-scanning settles at a particular ASTE for deactivation, the list-head pointer aste.level.ausedp is moved up to that ASTE, and after deactivation, to right ahead of it (as in the "some are free" case above). This tends to give the ASTEs skipped over in the scan a property of being "rejected for deactivation", and thus promoted to a less likely position to be seen next time, by virtue of this observation of "being recently used".

The replacement algorithm skips over ASTEs that cannot be deactivated; not only are these the ones with aste.ehs on (see the discussion of this flag above), but those with active inferior (directories who claim this ASTE as ASTE of their parent). All of the various reasons for skipping and moving on cause meters to be incremented, as well as file_system_meters (see the Multics System Metering PLM, Order No. AN52) that displays these statistics.

There is one circumnavigation of the required used list done before the "zero" pass: a preliminary "zero" pass is made that seeks segments with zero pages in main memory and the flag aste.init being off. This pass also turns off the flag aste.init when on, and all succeeding passes skip segments that have it on. Referring back to the description of aste.init, it is seen that this flag is turned on by page control when a segment acquires the property of having no pages in main memory. The effect of this policy is to allow segments that have zero pages in main memory to survive exactly one circumnavigation of the AST used list for that pool before being considered for replacement. This pass is the so-called "grace lap". It is an implementation of the policy: "if a segment just happens to have all of its pages float out of main memory, give it just one chance to get some back in before jumping on it to deactivate it." The file_system_meters command reports such skips as "skips init".

AST TRICKLE

Since the AST replacement algorithm is constantly inspecting all portions of the AST used lists, the opportunity is taken in that algorithm to notice ASTEs whose file maps have changed, and to update their VTOCEs at this time. This reduces the loop time of the AST replacement algorithm (reported as "grace time" by file_system_meters") to be a lower bound on the amount of time by which a VTOCE can be out of date. This is totally a hedge against fatal crashes; successful shutdown updates all VTOCEs of active segments. As mentioned before, this periodic update causes the physical volume salvager to notice certain incongruencies. Unfortunately, however, at times of light load, this lower bound is rather long.

LOCKING CONVENTIONS

There is one lock that protects the AST data base; it is called the "AST Lock", and is, in fact, sst.astl. It is a standard-format wait-type lock, managed by the procedure "lock". There are special entry points, lock$lock_ast and lock$unlock_ast to manipulate this lock, and limit knowledge of its location and format. The event for waiting on this lock is "400000000000"b3.

The AST lock has no cleanup mechanism; a crawlout with the AST lock locked (one is said to "have the AST locked" in this state), detected by verify_lock, or a process termination with the AST locked, crashes the system. The AST lock "protects" certain activities: this means that these activities may not be done unless the process attempting to perform them has the AST locked before commencing. These activities are:

1. Deactivation

2. Updating of VTOCEs (from the AST)

3. Manipulating the AST used lists, or following them, including the allocation and deallocation of ASTEs.

4. Using, following, or changing the AST hash table, and thus, determination of activity.

5. The calling of call-side page control entries on deactivatable segments.

6. Setfaults (see "Trailers and Setfaults" below).

The AST lock also protects against completion of the following activities: this is to say, these activities may be commenced by a process, but will not complete until that process has (holds, i.e., locked to that process) the AST lock.

1. Activation

2. Volume Demounting

The AST lock holds a position in the locking hierarchy above all directory locks and below wired locks as the traffic control and page control locks. It is below the VTOC buffer lock (see "VTOC Manager": "General Policies").

Since touching any nonsupervisor segment, such as a directory, can cause a segment fault, which would lock the AST, no directories or user-supplied supervisor arguments may be referenced by a process that holds the AST lock.

Note two major differences in the above policies from pre-4.0 locking policies:

1. The parent directory lock is no longer protection against deactivation of a segment.

2. Locked directories are not guaranteed to remain active, and thus cannot be locked by a process holding the AST lock.

The AST lock does not protect modification of VTOCEs. The directory lock of the directory containing the branch for the segment that owns a given VTCOE is the lock on that VTOCE if and only if the segment is not active. Since, when it is active, it may be deactivated at any time that a process seeking to deactivate it has the AST locked, the AST lock protects VTOCEs only when the segment owning the particular VTOCE is active. Thus, a procedure (such as vtoc_attributes) seeking to modify a VTOCE must perform the following protocol:

1. Lock the parent directory. If the segment is not active, it cannot become active while we hold the directory lock, for a directory lock fully protects activation of its inferiors. Procedures that wish to deal with segments and their VTOCEs in this way usually have the directory lock locked anyway.

2. Lock the AST lock. We cannot determine whether or not the segment is active without the AST locked, for not only is it not permissible to inspect the AST hash table without the AST locked, but lest the AST be locked to us, i.e., prevented from being locked by others, the segment might be deactivated at any time, or is being deactivated as we watch.

3. Determine if the segment is active. If it is, it may be sufficient to inspect or modify the activation attributes in the AST. Otherwise, in the case where the segment is active and dealing with the AST will not suffice, we must perform the modification while we have the AST locked, otherwise, another process might be trying to deactivate the segment, and thus engage in a simultaneous-update race with our process.

4. If we did not do so in step 3, unlock the AST and read and possibly change and write back the VTOCE. Since it was determined that the segment was not active in step 3, it cannot become active now, as we hold the parent directory lock, and this parent directory lock thus protects the VTOCE.

5. End of protocol; procedure may unlock the parent directory lock. See also "Services of Segment Control," in Section IV for utility of this behavior.

Note that in 4.0 and later systems, one can lock a directory without actually touching or inspecting the directory, simply by handing the directory's UID to the lock procedure. Thus, one can protect a VTOCE simply by inspecting its permanent information (vtoce.uid_path) to determine the UID of its parent, and handing this to the lock primitive. The procedure priv_delete_vtoce performs such machinations to delete orphans.

As mentioned above in passing, the lock on the parent directory of a segment totally protects activation of any segment; activation cannot <u>commence</u> until the activating process holds the parent directory lock.

There is a system of multiple-reader single-writer half-locks protecting against demounting; this is covered in Sections XIII and XIV.

<u>TRAILERS AND SETFAULTS</u>

One major feature of Multics is dynamic access control; as soon as a set_acl command is performed upon a segment, processes using the segment immediately take faults. This is implemented via the <u>trailer</u> mechanism, and the operations known as <u>setfaults</u>, implemented by the procedure of the same name.

Descriptor segments of processes contain SDWs. SDWs point to page tables, that reside in ASTEs. When ASTEs are replaced, all SDWs that point to that ASTE must be found, and faulted. Faulting an SDW consists of removing the bit sdw.df, and perhaps changing other information in the SDW. Setting this bit <u>off</u>, followed by a call to clear all the associative memories of the processors of the systems (privileged_mode_ut$cam) that might contain this SDW, causes the process attempting to use this SDW to take directed fault 0, which is known to Multics as a <u>segment</u> <u>fault</u>. Since this faulting is always done by deactivation, which has the AST locked, the process attempting to process the segment fault cannot determine whether or not the segment on which the fault was taken was even active until it can procure the AST lock, i.e., until the process doing the deactivating has fully deactivated the segment.

Since <u>all</u> SDWs pointing to a given segment must be <u>revoked</u> (faulted to be invalid) when a segment is being deactivated (or boundsfaulted on or segment-moved (see "Segment Moving", below), it is more efficient to keep a list of such SDWs, rather than search all of the descriptor segments in the system. This list is called the <u>trailer</u> <u>list</u> of the segment, and is stored in the segment (nondeciduous, paged, nonwired supervisor segment) str_seg. An entry in this list is described by the include file str.incl.pl1. Each entry consists of a forward thread to the next (zero if none), the AST offset of the ASTE for the descriptor segment of a process, and the segment number of the segment of whose ASTE this is the trailer, in that process. The ASTE field aste.strp gives the relative offset in str_seg of the first trailer entry of the trailer for the segment that owns the ASTE.

Trailer entries are threaded onto the front of the list for an ASTE each time the segment fault mechanism (in the procedure seg_fault) constructs an SDW (while protected by the AST lock). The manipulation or use of the trailer segment is protected by the AST lock. The SDWs constructed by segment-faulting upon deciduous segments in nonhardcore rings acquire trailer entries. The SDWs for deciduous (and all other hardcore and initialization segments) constructed by System Initialization do <u>not</u>, as they cannot be and are never revoked.

The trailer mechanism also locates all SDWs when an access change is performed upon an active segment (as via user command). This causes segment faults in all processes (see the description of "Segment Fault Handling" under "Services of Segment Control"). These segment faults will cause recalculation of access by these processes.

Needless to say, deletion of segments is a special case of the deactivation of active segments. This causes similar setfaults actions to be performed.

Setfaults are performed via the procedure "setfaults". The entry of greatest interest to segment control is setfaults$setfaults, which given an AST entry, "cuts the trailer", removing all trailer entries and revoking all SDWs. Setfaults also play a crucial role in encacheability management (see "Encacheability Control" below.) See also "Descriptor Segment Management" under "Service of Segment Control" for more about setfaults.

BOUNDSFAULTS

A boundsfault is the detection of a reference, by a process, to a word outside of the legal limit for the segment set in the SDW in that process. If outside of the maximum length of the segment (aste.msl), a boundsfault is signalled (the out_of_bounds condition). If not, this is simply a request to find a larger ASTE for the segment. This involves performing a "setfaults" on the old one, finding a new one, updating page control data bases (pc$move_page_table) and rethreading inferior father pointers. This operation is described in detail under "Services of Segment Control."

SEGMENT MOVING

It is possible for a segment to try to grow by a page when there are no more records available on the volume of its residence. If there is only one physical volume in the logical volume, this causes an error to be signalled (error_table_$logical_volume_full, as a subcondition of seg_fault_error). If however, there are other physical volumes in the logical volume, one of which has enough space to hold the grown segment, it is the system's responsibility to move that segment there transparently. This operation is known as segment moving, and involves a very complex interaction of page control and segment control, and is the most involved single service of segment control. Segment moving may also be performed on demand via the gate hphcs_, on behalf of the online pack utility sweep_pv, in order to vacate physical volumes (logical volume compression) and volume rebalancing. The details of this operation are given under "Services of Segment Control."

ENCACHEABILITY CONTROL

It would seem that the most appropriate place for the description of the policy used to manage the 68/80 cache is at this point.

The 68/80 cache is an associative memory of words from main memory in each 68/80 Multics processor. It is a write-through cache. That is to say, no word that the processor stores modifies a location in cache without modifying the encached location of main memory.

The fact that this cache is not transparent to the software, i.e., needs to be managed at all is a reflection of the fact that it is in the processors (for purpose of speed and modularity), and not in the 6000 SCU. Thus, words which a processor fetches from cache may have their copies on main memory modified by other processors, an IOM (or FNP6600 Communications Processor via the IOM), or a Bulk Store Subsystem, and the processor would not be able to observe these changes.

The Multics cache has a novel and powerful feature known as the encacheability of segments. This is to say that each Segment Descriptor Word (SDW) contains a bit (sdw.cache, bit 57) whose absence prohibits the processor port logic from loading words of that segment into the cache. Note that in absolute mode, where no SDW is used, all loaded words are eligible to be put in the cache. Thus, there are encacheable and non-encacheable segments, with sdw.cache "1"b and "0"b respectively. All SDWs used for a segment, be they created via segment faulting, or via initialization, must agree on encacheability.

For a start, all segments that are read or written by the IOM or bulk store for any reason other than paging, are nonencacheable. This includes a finite set of supervisor segments (e.g., tty_buf, dn355_mailbox, bulk_store_mailbox, iom_data, etc.), and all segments used as IOI Buffer segments (see "IOI Buffer Segments" under "Services of Segment Control" below). For the supervisor segments, the SDWs used are all created by initialization or copied from them.

Other supervisor segments are encacheable or not depending upon their "access". This "access" is the access that appears in all descriptor segments, developed from the one created by initialization for the initializer. Any segment with write access is not encacheable; all others are. Since segmentation restricts which segments are writeable at all, let alone by multiple processors, the only supervisor segments that are writeable at all are not encacheable. Thus, no supervisor segments may suffer the anomaly of being modified by one CPU while still visible in the cache of another. Two important exceptions to this rule are the PDS and PRDS created in the initializer process by initialization, and all KSTs, PDSs and PRDSs created thereafter. PRDSs (Processor data segments), after being initially created, are carried around by processors from process to process. After their creation, they are referenced by only one processor. Since only one processor can reference a given PRDS, it is encacheable; it is very important that it be encacheable, as it is used as a stack in wired and interrupt side ring zero. PDSs and KSTs are a special case of per-process segments, described immediately below.

Any segment may be encacheable if all of the SDWs describing it allow no write access (only read or execute). This has the same truth as for supervisor segments as above. However, if we take the same approach, we find that no writeable segments may be encacheable. This is unduly restrictive, for some writeable segments, such as stacks, linkage segments and KSTs, are among the most heavily used segments. It has been discovered that any segment accessible to only one process can be made encacheable if a simple rule is followed: any time a process switches processors (not the inverse), the new processor taking up that process must totally clear its cache. This specifically means that every processor as it switches to a new process need not necessarily clear its cache.

The proof of this theorem is as follows: assume a process P runs on CPU A, and some words of per-process segment X come into CPU A's cache. With no loss of generality, assume that CPU B has no words of segment X in its cache. As CPU A switches processes to and fro, there cannot be a problem until P runs on some other CPU, say B. This is because, by hypothesis, P has not run on B, and since it only has run on A, all words in A's cache are accurate, because the only process that can modify segment X, being P, has never run, by hypothesis, on any other CPU. When P finally runs on B, there is still no problem, because by hypothesis, CPU B's cache contains no words of segment X. Assume now that P modifies and fetches words from X liberally while running an B, specifically changing words that are still in A's cache. As long as P runs on B, whether or not other processes run in between runs of P, there is no problem, as these wrong words appear only in A's cache, and P is running only on B. When P is run the next time on A, the problem appears. There are words in A's cache that are inaccurate. The solution is simple: clear the entire cache of A. Thus, it is simple to do this every time when a process runs on a processor that is not the last one it ran on, clear the new processor's cache. This, of course, also fixes any potential problem when P transfers back to CPU B. Thus, are per-process segments like PDSs and KSTs encacheable. The traffic controller maintains the identity of the last processor on which a process ran, so the decision to clear the cache is easy.

The computation of encacheability for all nonhardcore segments is done in a uniform manner, in the procedure seg_fault. It will be seen that this policy allows per-process segments to be encacheable as a corollary.

Two bits in the AST entry of a segment describe one of four possible states with respect to the encacheability of the segment. Since only active segments have pages in main memory or SDWs describing them, only active segments are an issue. These states are:

1. No SDWs describe this segment. Its encacheability is not an issue.

2. One or more SDWs describe this segment. None of them allow write access. The segment is encacheable.

3. Only one SDW describes this segment. It allows write access. Since this is, at this time, a per-process segment by implication, as only one process can reference it, it is encacheable.

4. More than one SDW describes this segment, and at least one of them allows write access. The segment is not encacheable.

These bits are aste.any_access_on and aste.write_access_on. See the ASTE structure breakdown earlier for the correspondence between the states above and these bits.

All segments, when activated, are in state 1 above. Since only active segments have pages in main memory, the segment, when activated, has no pages in main memory. Page control clears out of all processor caches all words of a page being evicted from main memory (see Section VIII). Thus, a segment being activated has none of its words in any cache of the system, allowing the hypotheses of the preceding proof to be valid.

When any SDW, including the first, for an active segment, is created, the seg_fault procedure changes the encacheability state of the segment by modifying the two encacheability control bits in the ASTE of the segment. If it is moving from an encacheable state to a nonencacheable state, then setfaults$cache is called to revoke all of the cache bits in all of the SDWs that describe this segment, and cause an associative memory clear to force all processors to recognize this bit. This special setfaults entry does not revoke the SDWs, which would cause segment faults. This is not necessary here. The encacheable/nonencacheable status of the new SDW being added is derived from the encacheability status indicated in the ASTE.

When a system-wide setfaults is done, including a setfaults$cache, a clear of all processor's associative memories and caches is conducted by setfaults, by calling page$cam. When setfaults revokes all SDWs for a segment, therefore, it resets the cache state to state (1) above, for no SDWs describe the segment and no words of it appear in any processor's cache.

IOI Buffer segments, and the segment used to load the FNP6600 communications processor, cannot be encacheable, as stated above, even though they are only used in one process. Thus, at the time that they are force-activated, (see "IOI Buffer segments" in "Services of Segment Control") grab_aste_grab_aste_io sets the encacheability state to state 4 above, causing all SDWs constructed for the segment to specify nonencacheability, and sets aste.inhibit_cache on, whose sole purpose is to prevent setfaults from resetting the encacheability state when all SDWs are revoked (e.g., a set_acl was done on a buffer segment). This bit is reset by grab_aste$release_io.

Directories are not encacheable generally for historical reasons; they used to be addressable outside of the segment-fault-trailer mechanism, and thus were not subject to the policy above. Still, they are left nonencacheable, as it is felt that the referencing patterns of directories make it more desirable to not let them replace other segments in the cache, and thus ought to stay nonencacheable.

The encacheability attribute of hardcore segments is supplied by the MST generator; it is developed from the "access" and "cache" header statements. (See the Multics System Tools Reference Manual, Order No. AZ03.)

A limitation of the above encacheability policy is the lack of recalculation of encacheability as processes vanish or terminate segments, withdrawing their SDWs. It was felt that the class of segments that would benefit by such recalculation was small, and the overhead of being able to do this properly would be large.

SECTION III

THE VTOC MANAGER

INTRODUCTION AND OVERVIEW

Critical to the operation of Release 4.0 and all later systems is the concept of VTOC, the Volume Table of Contents, already detailed in the Segment Control Overview and Data Bases sections. VTOCEs are not part of the virtual memory, except when accessed by the physical volume salvager. This allows more efficient single-sector I/O to be performed on the VTOCEs. In order to make this I/O efficient, a buffering scheme for VTOCEs and their fractions must exist. This scheme is implemented by the VTOC manager, the procedure vtoc_man.

All VTOCEs are divided into three logical sections: the activation information, the file map, and the permanent information. A VTOCE may also be viewed as being divided into three physical parts, Part I, Part II, and Part III, as detailed earlier. Each physical subsection of a VTOCE comprising 64 words, is called a vtoce-part. The three vtoce-parts comprise the VTOCE.

All access to VTOCEs, other than that performed by the Physical Volume Salvager (and of course, BOS), is performed by calling entries in vtoc_man. The most general entries, vtoc_man$get_vtoce and vtoc_man$put_vtoce, read and write whole VTOCEs or single vtoce-parts. Other entries free a whole VTOCE (vtoc_man$free_vtoce), await completion of I/O on a VTOCE (vtoc_man$await_vtoce) and write a VTOCE to a free VTOCE, making it not free, and returning its VTOC index (vtoc_man$alloc_and_put_vtoce). There are also "global" entries to the VTOC manager that deal with no single VTOCE: vtoc_man$cleanup_pv, called at volume demount and shutdown time (see Section VM), and vtoc_man$stabilize, called at ESD time to ensure consistency in the state of the VTOC manager's data base.

The VTOC manager uses the segment vtoc_buffer_seg as a data base, containing all variables needed in VTOC management, which are not global parameters to a given volume. Many of the variables in the Physical Volume Table, (PVT), such as the heads of VTOCE free chains, and number of free VTOCEs, are for use by the VTOC manager. The VTOC buffer segment, vtoc_buffer_seg, contains up to sixty-four vtoce-part buffers. Each buffer, 64 words long, is either free or contains one vtoce-part. Vtoce-parts may be from any mounted physical volume, and no two buffers contain the same vtoce-part. There is no free list of any kind. Thus, any vtoce-part of a mounted volume is either in exactly one vtoce-part buffer or not in any. Note that a vtoce-part buffer containing a vtoce-part of a free VTOCE is not a free vtoce-part buffer; the latter is one that contains no vtoce-part of any VTOCE.

There is a table in the VTOC buffer segment containing single word buffer descriptors, also known as buffer control words. Each describes the status of one vtoce-part buffer, stating which part of which VTOCE if any is contained there, and other status information. The format of this control word is described later.

It is the goal of the VTOC manager to provide interface to VTOCEs, for segment control programs, without these programs being aware of the buffers, their existence or their organization. The VTOC manager must implement a buffer multiplexing, and therefore, a sharing algorithm. The VTOC manager is unaware of the content of VTOCEs, other than the manipulation and maintenance of the VTOC free thread. It is also the responsibility of the VTOC manager to interface to the disk control software to actually perform the VTOC I/O.

GENERAL POLICIES

The VTOC manager, at its lowest level, manages vtoce-parts and their buffering. At any given entry to the VTOC manager, the vtoc buffer segment contains a given set of vtoce-parts: in order to satisfy a request for most calls, the requested set of vtoce-parts are either among the set in the buffers in part, in whole, or not at all. If they are all there, this data may be used or returned without any I/O. If the requested vtoce-parts are in part or in whole not in the buffers, they must be brought in.

Searches and replacements of vtoce-part buffers are protected by the VTOC Buffer Lock. This lock is standard-format wait-lock, managed by the locking procedure "lock." Its notify event is "3330000xxxxx"b3, where xxxxx is one greater than the number of vtoce-part buffers. It is higher than the AST lock. When the VTOC manager waits for I/O, it unlocks this lock so as not to tie up this resource. Therefore, vtoce-parts that were present when this I/O was started may not be present when the I/O is complete, for operations involving more than one vtoce-part. This situation is analogous to the paging behavior of multi-operand EIS decimal instructions: they continue to fault, with no assurance that they will be satisfied in any given time constraint, until all pages are found present at once.

The policy of getting together all buffers at once (implemented via the internal routines GET_BUFFERS_READ and GET_BUFFERS_WRITE described below) is the implementation of a design constraint that all calls to the VTOC manager be unitary operations with respect to volume demounting. This is to say, when modifying VTOCEs, a call to the VTOC manager will cause either all requested vtoce-parts to be modified as needed or none, given a volume demounting at any stage of the operation. This policy allows procedures such as vtoc_attributes to read VTOCEs and write them back via only two calls to the VTOC manager, the second call either wholly succeeding or wholly failing. Thus, such a procedure need not be explicitly protected against demounting. (See Section XIV for a discussion of Demount Protection.)

Furthermore, operations to modify vtoce-parts, which write them wholesale (the VTOC manager does not modify or inspect parts of vtoce-parts), must use the buffers occupied by these vtoce-parts if there are any; were this not the case, some vtoce-parts would have more than one buffer associated with them, and a question of relative legitimacy would arise, as well as issues of multiple I/O operations on a given vtoce-part at once. Thus, this policy of only one buffer per vtoce-part assures not only a finite small set of buffer states, but a similar small set of states of any vtoce-part in the system with respect to the VTOC manager.

The VTOC manager receives requests in terms of VTOCEs, with masks specifying which vtoce-parts are being dealt with, in the "get" and "put" entries, as well as pointers to data areas to copy to and from. The specification of a VTOCE is via a PVT index (the PVT is the Physical Volume Table, the table of all mounted physical volumes) and a VTOC index. The circumstances under which Physical Volume Table indices may validly be derived and used are given in Section XIV of this document. It is part of that protocol that no volume demount may complete while the demounting process does not have the VTOC buffer lock locked. Therefore, the VTOC manager is protected against demounting. However, procedures that call the VTOC manager are not protected against demounting. Therefore, the PVID (physical volume ID) of the volume that the caller expects to be dealing with is passed as an argument to the VTOC manager. If, while the VTOC buffer lock is locked, the supplied PVT index indeed checks with this PVID (by inspecting the PVT), all is well. Every time the VTOC buffer lock is relocked, this check must be made. If it does not, the caller is informed that the volume being referenced was demounted (err_table_$pvid_not_found). If this parameter is passed as "0"b, it means that the caller has some other protection against demounting such as having the AST locked.

The procedure vtoc_interrupt is the interrupt side of the VTOC manager. It is called from the disk DIM at any time that the disk DIM processes status. This procedure does <u>not</u> lock the VTOC buffer lock. As vtoc_interrupt is called in a wired, masked environment, in which the running process may even have the global page table lock set (see Section XIII), were it to lock the VTOC buffer lock, that would mean that all procedures that lock this lock, notably vtoc_man, would have to run in masked, wired environments, which are expensive to obtain. Thus, the interrupt side of the VTOC manager runs asynchronously. This procedure modifies bits in the VTOC buffer control words, specifically b.os and b.err, completely asynchronously. The rest of the VTOC manager must be prepared for these bits to change for any buffer for which I/O is in progress, at any time.

Every call to the VTOC manager, other than the global call vtoc_man$stabilize, deals with one specific mounted physical volume. A variable is kept in the VTOC buffer segment, vtoc_buffer.unsafe_pvtx, which designates a physical volume being processed. Should the system crash, ESD will inspect this field and schedule that volume for volume salvage (see Section XIV).

The individual procedures and entry points of the VTOC manager are clearly documented in the program listing. Thus, we now provide a detailed breakdown of the data structures of the VTOC manager, being the VTOC buffer segment and the buffer control words therein, and describe after that the basic subroutinization strategy of the program vtoc_man.

<u>VTOC BUFFER SEGMENT</u>

lock

is the VTOC buffer lock. It is a standard format wait-lock, whose event ID is stored in vtoc_buffer.lock.ind.

n_buffers

is the number of vtoce-part buffers in the VTOC buffer segment. It is computed by init_vtoc_man, from a parameter on the PARM VTB CONFIG card.

abs_add

is the absolute address of the base (word 0) of the VTOC buffer segment. It is contiguous in main memory. This allows the VTOC manager to compute the absolute address of each buffer for calls to the disk DIM.

event_constant

is a constant from which all VTOC buffer wait events are constructed. This constant is "333000000000"b3. The wait event for the completion of I/O in buffer number <u>n</u> is vtoc_buffer.event_constant + n. For example, the wait event for awaiting I/O on buffer 5 is "333000000005"b3.

current

is the current replacement pointer, a buffer index. See "VTOC Buffer Replacement Algorithm" below.

unsafe_pvtx

is the index in the Physical Volume Table (PVT) of the single physical volume on which operations are in progress when the VTOC buffer lock is locked on behalf of an operation on a specific volume. It is inspected by Emergency Shutdown to schedule a salvage for that volume if found nonzero. It is cleared when the VTOC buffer lock is unlocked.

inhibit_await

is for debugging use only. When nonzero, it inhibits the feature of awaiting successful completion of VTOC I/O before addresses are deposited (the function performed by vtoc_man$await_vtoce). This feature is critical to the address management policy of Multics (see "Address Management Policy" in Section VII).

mtr

is a group of meters, most of which are printed out by the vtoc_buffer_meters tool. Of particular interest are vtoc_buffer.mtr.parts_read and vtoc_buffer.mtr.parts_write, which are distributions of read and write requests, indexed by combinations of vtoce-part.

Description of the VTOC Buffer Control Word, vtoc buffer.b

b.used

indicates whether or not this buffer contains a vtoce-part. If b.used is "0"b, no other bits in the buffer control word are valid.

b.os

for "out-of-service" indicates that I/O has been queued for this buffer, and has not been posted (completed). This bit is turned on by vtoc_man prior to calling the disk DIM, and turned off only by vtoc_interrupt, asynchronously (and by vtoc_man$stabilize, called only at ESD time). This bit and b.err, below, are the only two bits managed asynchronously. As in page control, "out-of-service" means "I/O in progress", not "damaged" or "unusable".

b.op

indicates the last operation, or the one in progrpss, on this buffer. Zero is read, one is write.

b.waitsw

tells whether or not (1 equals "yes") some process is waiting for I/O complete on this buffer. If on, vtoc_interrupt will call the traffic controller to notify the event constructed as described under vtoc_buffer.event_constant. This bit also prejudices the replacement algorithm (See "VTOC Buffer Replacement Algorithm", below) against this buffer.

b.ioq

Is set to "on" after a request for I/O has been queued. This is used to reduce uncertainty about whether or not I/O completion will be posted at ESD time. Any buffer encountered at ESD time with both b.os and b.ioq on can expect a completion posting from the disk DIM. See the "VTOC Manager ESD Strategy" description below.

b.err

is set on asynchronously by vtoc_interrupt, at buffer I/O completion time if this I/O completed with an error. When found on for a read operation, the process that was waiting for this read to complete notices this and returns error_table_$vtoc_io_err out of vtoc_man, and frees the buffer, as it contains no good vtoce-part. For a write request, the vtoce-part becomes "hot": this is to say that it is known that this buffer must be updated to disk at some later time, for the VTOCE on disk is known not to have these modifications. See "Error Strategy" below.

b.partno
tells which vtoce-part of a VTOCE, 01, 10, or 11, resides here.

b.pvtx
is the PVT index of the mounted physical volume to which this vtoce-part belongs.

b.vtocx
is the VTOC index of the VTOCE, in the VTOC of the mounted physical volume to which this vtoce-part belongs.

There are also two internal static variables of vtoc_man: alloc_state and select_state. These are pseudoclocks that are advanced whenever an allocation or VTOC buffer selection, respectively, is performed. By saving and comparing these values to their saved values, vtoc_man is able to determine whether or not these operations have occurred during an unlocking of the VTOC buffer lock.

ORGANIZATION OF THE VTOC MANAGER

The structuring of the VTOC manager must be comprehended in order to understand and diagnose problems and changes in this area. A listing of vtoc_man should be on hand to best follow this section.

The critical intermediate level subroutines are the two named GET_BUFFERS_READ and GET_BUFFERS_WRITE. These subroutines receive the specification of the VTOCE to be dealt with (PVT index and VTOC index) via global program variables: a three-bit vtoce-part mask is passed as an argument, as is a return code. The function of both of these procedures is to associate one to three buffers with the requested vtoce-parts. For reading, (GET_BUFFERS_READ) this includes performing (initiating and completing) I/O to read in these vtoce-parts if they are not already in the VTOC buffer segment. For writing, this means finding buffers containing any of the requested vtoce-parts, if any, and allocating new buffers for those not already in the VTOC buffer segment. In both cases, these routines return the indices of the found/filled/allocated buffers via the array "A", being in A(1), A(2), A(3) for the respective vtoce-parts, when requested. In both cases, the routines are responsible for performing these operations consistently, which means observing changes that happen during unlocking, and retrying the buffer-gathering when necessary (see the "General Policies" discussion earlier).

These two primitives are very powerful; the implementation of vtoc_man$get_vtoce is little more than a call to GET_BUFFERS_READ. The implementation of vtoc_man$put_vtoce is little more than a call to GET_BUFFERS_WRITE, copying of the data supplied into these buffers, and calls to the WRITE subroutine to start I/O on those vtoce-parts. Thus we proceed to discuss the operation of GET_BUFFERS_READ and GET_BUFFERS_WRITE.

Both routines start by establishing a retry point. If any operation causes an unlocking, and subsequent relocking shows that buffers involved in this operation have been replaced, the operation is restarted from this retry point (label START in both routines.) Both routines then call the subroutine INIT, to fill up the array A with either -1 (vtoce-part wanted, not yet found) or gotten or -2 (vtoce-part not even wanted), and get the minimum and maximum part number out of 1, 2, and 3. The routine SEARCH is now called to scan the VTOC buffers to fill in "A" with the indices of all found vtoce-parts (that are needed) of this VTOCE. The value returned by this routine is the number of vtoce-parts found. At this point, GET_BUFFERS_READ and GET_BUFFERS_WRITE differ. GET_BUFFERS_READ proceeds by first selecting a new buffer and then starting a read (subroutine READ) for each vtoce-part wanted but not found by SEARCH. The buffer selector, SELECT_BUFFER, which implements the buffer replacement algorithm, is careful not to disturb buffers already pointed at by "A". GET_BUFFERS_READ then calls WAIT, to wait for any of the gotten buffers which were, or are now, out-of-service (I/O in progress). Since this waiting (performed by calling WAIT_OS on each out-of-service buffer) may unlock the buffers, it is necessary to check that each buffer described by "A" still contains the vtoce-part it did when put in A. This check is performed by the routine "VANISHED", which makes precisely this check. A branch to the retry point START is performed if it fails. This check is bypassed if it is determined that the select pseudoclock (see above) has not moved during the unlocking. The WAIT routine is intelligent about seeing that <u>all</u> buffers in A are not out-of-service when it returns.

GET_BUFFERS_WRITE, having searched for all relevant vtoce-parts, proceeds by calling WAIT so that they are no longer out-of-service. While this waiting is not strictly necessary in the write case, it is a very conservative action. At the end of this operation, the check for "VANISHED" and conditional branch back to the retry point are undertaken. Then the selector, which is careful about not disturbing buffers described by A, is called to get buffers to associate with those vtoce-parts that were <u>not</u> found by SEARCH.

All the rest of the subroutines are basically support for GET_BUFFERS_READ and GET_BUFFERS_WRITE: these and the few other subroutines will be described below.

csyser

subroutine to crash system by calling syserr. It exists in order to common code printing out drive identification, and set vtoc_buffer.unsafe_pvtx to schedule a volume salvage.

CHECK_PART3

a debugging subroutine that checks the third vtoce-part for reasonability. From times when there were problems in this area.

WAIT_OS

A lowest-level primitive to wait for the buffer specified by its first argument to stop being out-of-service. This subroutine concerns itself with the traffic controller wait-retry-addevent protocol, and the locking and unlocking of the VTOC buffer lock around real waiting. The event for which it waits is described under the description of vtoc_buffer.event_constant. The code returned is that returned by LOCK_BUFFERS, if nonzero. See that description below.

LOCK_BUFFERS

calls the system lock primitive lock$lock_fast to lock the VTOC buffer lock. It also checks, upon every relocking, that the PVID supplied by the caller of vtoc_man still corresponds to the PVT index given, and that a demount has not started, nor the drive become inoperative. The occurrence of these conditions is reflected in LOCK_BUFFERS' return code.

UNLOCK_BUFFERS

unlocks the VTOC buffer lock, using the system unlocking primitive, lock$unlock_fast.

VANISHED

Described above. Scans the array A to see if the buffers described by "A" still contain the vtoce-parts of the VTOCE being processed (in the right order), after an unlocking during which the select pseudoclock has moved.

INIT

described above, initialized the array "A" for GET_BUFFERS_READ and GET_BUFFERS_WRITE. -1 is wanted but not yet found or got, -2 is not even wanted.

WAIT

calls WAIT_OS for each vtoce-part in a VTOCE being processed that is out-of-service. Returns only when none are out-of-service.

SEARCH

Fills up the array A with buffer indices for all vtoce-parts needed, by searching the VTOC buffer segment for all vtoce-parts that are there already.

READ and WRITE

Given the vtoce-part number (part number) these routines actually call disk control to start I/O on the vtoce-part and buffer. These routines set up the buffer control words, placing b.os (I/O in progress) on, and b.ioq on after the return from the call to disk control.

RECORD, SECTOR and CORE

are used by READ and WRITE to convert VTOC indices into Multics record number and sector within that record (taking the particular vtoce-part into account), and to get the absolute main memory address (see description of vtoc_buffer.abs_address.)

VERIFY_ERROR_FREE

is used by the vtoc_man$await_vtoce entry to wait for all vtoce-parts of a given VTOCE to complete their I/Os, and report whether or not all of these I/Os were successful. The successful completion of the I/O for a write is a necessary prerequisite for address deposition (see "Address Management Policy" in Section VII, and "Segment Truncation" under "Segment Control Services").

SELECT_BUFFER

is used to obtain a new buffer for GET_BUFFERS_READ or GET_BUFFERS_WRITE when a requested vtoce-part is not already in the VTOC buffer segment. It gets a new one by replacing an old one. It does not unlock the VTOC buffer lock in any case. In replacing an old one, it implements the VTOC buffer replacement strategy described below.

VTOC BUFFER REPLACEMENT STRATEGY

Free vtoce-part buffers are needed by GET_BUFFERS_READ and GET_BUFFERS_WRITE when not all requested vtoce-parts are found in the VTOC buffer segment. The routine SELECT_BUFFER in vtoc_man allocates buffers in an essentially FIFO manner. A circulating pointer (vtoc_buffer.current) marks the next point to be inspected for replacement, behind this being the last one allocated. Buffers are allocated by circumnavigating the buffer segment a very large number of times, if necessary, until a buffer is found which is not out-of-service or "hot" (see "Error Strategy" below), and is not a vtoce-part of the VTOCE for whom buffers are being sought. (This prevents it from undoing its previous work by accident). Unused buffers fall into this category, as well as just ordinary buffers that meet these criteria. The first pass around the buffers, in a given call to SELECT_BUFFER, buffers with b.waitsw are skipped. These are buffers on which I/O was completed (remember, b.os was found off), and processes have been notified for, and will use when they get the VTOC buffer lock. Since these are only preempted in a bad case (second pass), this is not a performance problem. The process which comes back will find that the primitive "VANISHED" is now true, and will retry its buffer-gathering.

The pointer vtoc_buffer.current is advanced as each new buffer is allocated. When a very large number of passes over the VTOC buffer segment have failed, system operation is terminated. Note that the longer one scans, the more I/O operations complete, and buffers become claimable.

ERROR STRATEGY

We speak here of the "errors" encountered by the VTOC manager as a result of I/O operations completing with an error (b.err is on). The expectable "errors" of volumes being demounted or buffers vanishing are not errors at all, but designed features, and have been covered.

Disk errors can occur on reads and on writes, the only two operations performed by the VTOC manager. The strategy for a failing VTOC read is simple. If the buffer has not vanished by the time the process (or any process) which wanted to read it, this process notices the error (b.err is on), frees the buffer (so that the next call will not find it here, as it does not contain the vtoce-part it is supposed to, and so that the next call retries the operation), and returns error_table_$vtoc_io_err to its caller.

Write errors are substantially more difficult. In general, the completion of a write operation is not waited for by any process, and there is thus in general no process that can be relied upon to process the buffer in error. When a buffer is posted with a write error (vtoc_interrupt issues a syserr message in this case), the buffer concerned enters a state called "hot" (a hot buffer). It is so called, when b.op = b.err = "1"b, because the vtoce-part in it must be written to disk at some time before the system is shut down or the volume demounted, and if it cannot be, the volume must be salvaged before ever being mounted again. Furthermore, the "hot" buffer cannot be replaced, because it is the only valid copy of that vtoce-part, because, by hypothesis, we could not write it to disk. Thus, all calls to GET_BUFFERS_READ or GET_BUFFERS_WRITE must find the vtoce-part in this buffer. This buffer may not be replaced, so that vtoc_man$await_vtoce will find that the writes that were requested via vtoc_man$put_vtoce have failed, and so that the caller will know in this case that the VTOCE was not successfully written to disk. In this case, the usual callers (truncate_vtoce, update_vtoce, etc.) must not deposit addresses culled from the file map, for should the system crash before the VTOCE is written out, those addresses find their way into some other VTOCE, and a reused address results. (See "Address Management Policy" in Section VII, and "Segment Truncation" under "Services of Segment Control," Section IV.)

Every time some new caller of vtoc_man tries to issue a write on that buffer, the error bit is turned off, and may or may not be turned on depending on whether the operation succeeds, or fails again. Thus, each attempt to do a put_vtoce on that vtoce-part retries the failing operation, until successful.

One last try to write out all hot buffers is made at volume demount time (regular or emergency shutdown is effectively demount time for all volumes mounted then). If this last try fails, the disk being demounted is scheduled for salvage the next time it is mounted. This operation is performed in vtoc_man$cleanup_pv.

ESD STRATEGY

The basic problem of the VTOC manager at ESD time is to restart all I/O for buffers that are marked out-of-service, but for which the disk DIM does not currently have I/O under way. Since there is no way to determine this by interrogating the disk DIM, heuristics are used. The idea is to restart those and only those operations that are in this indeterminable state. If I/O is requeued for a buffer for which the disk DIM later posts completion, a double posting and double I/O, reading or writing of the wrong data will happen.

This would be detected by vtoc_interrupt when a buffer was not out-of-service received an I/O completion. On the other hand, if we do not start I/O for a buffer for which I/O was not actually pendent in the disk DIM, we would wait forever for its completion. Since b.os being on the b.ioq being off identify all buffers in this uncertain state, if there are any, a wait of thirty-five seconds is performed, for the disk DIM to post it if it is ever going to be posted. If it is not posted in this time, it is declared not-to-be out-of-service, and the I/O is requeued.

Emergency shutdown, as all shutdown, flushes "hot" buffers as described under "Error Strategy" above.

VTOCE ALLOCATION/DEALLOCATION SERVICE OF VTOC MANAGER

The VTOC manager is responsible for allocating and deallocating VTOCEs upon request. As mentioned before, a free chain of actual free VTOCEs on each volume is kept threaded through them, the head of the chain being in the PVT entry for that volume.

Deallocating VTOCEs is rather simple: a vtoce-part of zeros, with a free thread logically replaces the first vtoce-part of the VTOCE being freed. The VTOC index of this VTOCE becomes the new head of the chain in the PVT. GET_BUFFERS_WRITE is used herein. Allocating is more complicated. It is necessary to read the VTOCE that is designated as the head of the free chain in order to get the next free chain head. Since a waiting (with consequent unlocking of the VTOC buffers, must be performed to do this, it is possible that another process can attempt to allocate the same VTOCE as this process is allocating. This is because the PVT chain head cannot be changed until this VTOCE has been (first vtoce-part thereof) read in. Thus, the pseudoclock "alloc_state" is used every time this first phase of allocation is undertaken. If, upon relocking, an allocating process notices that this clock has moved, the operation is restarted. The nonmoving of the pseudoclock signifies that no other process has attempted to allocate that VTOCE during the unlocking. The entry vtoc_man$alloc_and_put_vtoce writes the new contents of the VTOCE out, once it has succeeded in allocating it. This protects the allocate-and-put primitive from demounting: if it got as far as changing the PVT thread head (actually performed the allocation), it actually started the writes. The writes being in progress (b.os is on) when the VTOC buffers are unlocked prevent the volume from demounting until the writes are complete (see Section XIV). The routines GET_BUFFERS_READ and GET_BUFFERS_WRITE are both used to fullest advantage in the allocate-and-put primitive.

SERVICES OF VTOC MANAGER FOR DEMOUNTING

When a volume is being demounted (recall that both normal and emergency shutdown are special cases of volume demounting for the entire mounted hierarchy), vtoc_man$cleanup_pv is invoked on behalf of that volume as one of the last stages of demounting. (See Section XIV). The vtoc_man routine makes a final try at outputting all "hot" buffers. Then vtoc_man waits for all VTOC I/O on the volume to cease; it has been guaranteed that no more can start by the setting of the bit pvte.demounting2 by demount_pv. (This bit is inspected by all attempts to lock the VTOC buffers: see the description of LOCK_BUFFERS above). No more VTOC I/O transpires on this volume; the VTOC is updated and quiescent. All vtoce-part buffers that had contained vtoce-parts of the demounted volume are marked as empty (free).

SECTION IV

SERVICES OF SEGMENT CONTROL

This section describes the meaning, organization, and implementation of the services provided by segment control to Multics. These are the functions that segment control performs; its reason for being. These services are built upon the mechanisms and data structures described earlier in this section.

These are the basic services of segment control:

1. Creating segments.
2. Destroying (deleting) segments.
3. Truncating segments.
4. Making segments addressable by processes (satisfying segment faults). This involves activation and deactivation as described.
5. Descriptor segment management.
6. Handling boundsfaults.
7. Setting and reporting "VTOC attributes" of segments.

These are the auxiliary services of segment control:

1. Special-casing per-process hardcore segments (PDSs and KSTs) with forced activations and special address management policies.
2. Special-casing of IOI buffer and FNP6600 Communications Processor bootloading segments.
3. Performing segment moving, both on demand and in response to physical volume overflows.
4. Performing special services on behalf of the online pack utility, sweep_pv, such as anonymous VTOCE deletion.
5. Supporting the hierarchy salvager.
6. Demand deactivation.
7. Shutting down segment control.

The segments above are only segments in the storage system hierarchy; the nondeciduous hardcore segments, PRDSs and descriptor segments are created by means external to segment control (see the Multics Reconfiguration and Multics Initialization PLMs, Order Numbers AN71 and AN70), and are dealt with by other parts of the supervisor by direct interaction with page control. Such segments have neither branches nor VTOCEs, do not count against any record quota, and are never activated or deactivated or in any AST list, hash thread, or father-son-brother chain.

Many of the top-level services of segment control (creation, truncation, deletion) are performed by similarly-named procedures (create_vtoce, truncate_vtoce, and delete_vtoce) in bound_vtoc_man. These deceptively named procedures do not in general perform operations upon VTOCEs, but either upon VTOCEs, AST entries, or some combination of the two, usually by calling page control primitives when operations upon ASTEs are required. It is these procedures that decide where the appropriate data about the segment being dealt with lies, and call appropriate entries to the VTOC manager when necessary. These procedures are called by the directory control programs append, truncate, and delentry, which create and delete directory branches, and check access and locate branches in all cases. Thus, create_, truncate_, and delete_vtoce should be thought of as create_, truncate_, and delete_segment.

The procedure vtoc_attributes falls right into this model, as an intermediary between the directory control primitives "set" and "status", setting or returning the so-called VTOC attributes in either the ASTE or VTOCE as necessary.

All of these primitives are called with the parent directory of the segment under consideration locked.

Among the descriptions of the services provided by segment control will be found a description of the VTOC update function, update_vtoce. While this function is entirely organizational, an artifact of implementation rather than of services, its critical role in the segment control panorama requires that it be described in detail in this section.

CREATION OF SEGMENTS

Creation of segments is performed via creating VTOCEs for them, by the procedure create_vtoce. The input parameter to this program is a complete directory branch. The principal output parameters are a physical volume ID (PVID) and VTOC index of a VTOCE that was created. The VTOCE creation function is called both by append (normal creation of segments) and the segment mover, segment_mover (See the detailed description later on in this discussion of Segment Moving).

The principal goals of VTOCE creation, as performed by create_vtoce, are these:

1. Create a local image of the VTOCE to be created. Fill in UID, primary name, VTOCE permanent information, initial values of activation information, a null (all pages null addresses) file map. Determine the UID path and fill that in too.

2. Find an appropriate physical volume for residence of the new segment. This must be one of the physical volumes of the logical volume that is the sons_lvid of the directory in which the given branch appears. Special case the rpv_only directory, ">lv". Select the most appropriate physical volume, as described below under "PV Selection Algorithm". (See Section XIV for motivation for this policy.)

3. Invoke the VTOC manager (vtoc_man$alloc_and_put_vtoce) to allocate a VTOCE on a selected physical volume, and write out the VTOCE constructed in step 1 to it. Receive back the VTOC index of the VTOC chosen by the VTOC manager.

4. Return to PVID of the physical volume selected by step 2 and the VTOC index of the VTOCE selected by step 3 to the caller, who usually places them in the branch (entry.pvid and entry.vtocx).

This function is not protected against demounting of volumes. However, nothing it does until the call of vtoc_man$alloc_and_put_vtoce has any side effect. Thus, should the call to vtoc_man fail because of demounting, create_vtoce will simply go back, select another physical volume and retry, until either no more physical volumes that are acceptable are left, or the logical volume becomes unavailable.

When operating on behalf of the segment mover, create_vtoce does not consider all physical volumes in the logical volume as potential candidates for the new VTOCE, but only those not yet inspected during this segment move. (See "Segment Moving", later in this section.)

Physical Volume Selection Algorithm

This algorithm is used by create_vtoce to find an appropriate volume for a new VTOCE, and thus segment, being created. Its main goal is to try, when not being invoked on behalf of the segment mover, to optimize balancing segments over the physical volumes of a logical volume, without causing undue I/O contention by placing many new segments in the same place.

The algorithm is to walk the chain (through pvte.brother_pvtx) of mounted physical volumes of a mounted logical volume. The head of this chain is kept in the logical volume table (LVT) (See Section XIII of this document for more details on these data bases.) In the case of the segment mover, this chain is walked from the last point it was at during this segment move until any acceptable physical volume is found; in the normal case, the whole chain is walked until the "optimal" physical volume is found. No physical volume is acceptable in any case if it is "vacating" (pvte.vacating is on, signifying that sweep_pv is trying to vacate this volume, or inhibit creation upon it), or has no free records left (records left is recorded and maintained by page control in the PVT entry). For segments that must be on the RPV (sons of the ROOT directory (>) or sons of >lv), no volume but the RPV is acceptable. The optimal physical volume, for all cases except per-process segments, is that which has the highest <u>percentage</u> of space available, in terms of unused paging records. This criterion, rather than absolute amount of paging space available, allows different capacity packs to be put in the same logical volume and fill up uniformly.

Per-process segments, those with entry.per_process in their branches, are dealt with differently. This is because these segments see heavy use, and the policy used above for other segments would place many new per process segments in the same place, such as a new pack added to a logical volume, causing a severe I/O bottleneck on that pack. Thus, a rotating pointer through the logical volume chain, lvte.cycle_pvtx is maintained by create_vtoce, pointing to the next Physical Volume in the round robin that will receive the next segment creation in that logical volume. The other acceptability criteria are still used; rpv-only creations, those on behalf of the mover, and those for which this round robin technique causes detectable looping (volumes seem to become unacceptable as they are inspected) cause the non-per-process algorithm to be defaulted to.

The significance of zero in lvte.cycle_pvtx is that it has either never been used, or has cycled around to the end of the chain.

The create_vtoce procedure operates with the knowledge that neither the logical volume table nor the PVT thread are protected by locks, and therefore, treats these quantities as asynchronously variable.

DELETION OF SEGMENTS

Deletion of segments, at the segment control level, is performed by the procedure delete_vtoce. The input parameter to this procedure is a directory branch (this implies that the directory in which it resides is locked to this process). There are no output parameters, other than the obligatory status code. The segment deletion function is called from the directory control program "delentry", which resolves pathname or segment number references to segments to be deleted, locates the branch for the segment, and checks that the caller's access is adequate to perform this deletion.

Deletion at the segment control level consists of the following main steps:

1. Make the segment inaccessible, if active, via a setfaults. Recall that the parent directory is locked, and segment faults on this segment cannot be processed by other processes until this process releases the parent directory lock. The entry setfaults$if_active performs exactly the flavor of setfaults needed here.

2. Truncate the segment to zero-length. The procedure truncate_vtoce comes right into play here, almost exactly as if called by the directory control truncate primitive. This releases all disk, bulk store, and main memory pages occupied by the segment. No more can be created, since all SDWs were revoked in Step 1, and the segment is inaccessible.

3. If this is a directory with a quota account being deleted, call the page control quota move primitive, quotaw$mq, to relinquish its quota to its superior. If this is any kind of a directory being deleted, directory control has already made sure that there are no segment or directory branches in this directory, so it has no inferiors, or inferior segments which might count against quota.

4. If this segment is active, deactivate it. This releases its ASTE. All pages of the segment were released in Step 2.

5. Free the VTOCE with a call to vtoc_man$free_vtoce.

Among the fine points of delete_vtoce:

This procedure, as described, is not protected against volume demounting. Thus, were a volume on which delete operation were under way demounted while the delete operation was between steps 2 and 5, a truncated segment would appear the next time this pack were mounted: whereas we desire either the original segment, or the lack of a segment. Thus, for multistep operations such as VTOCE deletion, a form of demounting protection known as "demount protection brackets", described fully in Section XIV of this document, was developed. Basically, a call to get_pvtx$hold_pvtx before step 1 prevents the volume from being demounted, or returns the fact that it has already been demounted, before step 1 above even begins. A call to get_pvtx$release_pvtx after step 5 releases the volume for demounting. See Section XIV of this document to find out what happens when a crawlout, process termination or crash happens while a process has such a grip on a volume. Since truncate_vtoce normally also makes such calls, a special entry to truncate_vtoce (truncate_vtoce$truncate_vtoce_delete) is used, which avoids making such calls knowing that delete_vtoce is doing it instead.

The program truncate_vtoce is capable of indicating a connection failure: this is to say the VTOCE designated by the PVID and VTOCX in the branch is either free or contains a UID other than the one in the supplied branch. In this case, delete_vtoce wryly notes that it is being asked to delete something which has clearly vanished of its own accord (can happen in crashes; the Physical Volume Salvager also sometimes creates this situation deliberately), buries the error, and returns indicating successful completion (after releasing the physical volume for demounting, of course).

SEGMENT TRUNCATION

Truncation of segments is performed by the procedure truncate_vtoce. This procedure is invoked both by the directory-control program "truncate", which converts pathname and segment number references to segments to be truncated into branch pointers, and checks appropriate access, and the segment deletion primitive already described. The inputs to this procedure are a branch pointer (with the directory of course locked) and a page number from which to start truncating. For the delete case, this number is assumed zero. The only output parameter is the error code.

Truncation may be defined as associating logical zeros with the contents of all pages beyond a certain point in a segment. For active segments, this is done by the page control primitive pc$truncate (which can also be used on nonstorage-system-hierarchy segments). For nonactive segments, it is done by freeing nonnull record addresses in the VTOCE file map, and replacing them with null device addresses.

Among the major issues in truncation is the implementation of the address management policy as described in Section VII of this document. The repercussion here is that record addresses may not be <u>deposited</u> (placed in the free storage pool for that pack, by calling pc$deposit_list) until it is known for a fact that the VTOCE from which they were removed has been <u>successfully</u> written out to disk. Were this not so, it would be possible that some addresses might be deposited, picked up by a new segment, and written out to that VTOCE. Then, if the VTOCE which <u>had</u> the addresses originally was not yet successfully written out, or badly written out, and the system crashed at that point, two VTOCEs would both contain the same record address, a situation known as a "reused address" which is a bad security violation. Thus, the primitive in the VTOC manager, vtoc_man$await_vtoce, is provided for just the purpose of waiting for successful I/O completion on the writing of VTOCEs.

Another issue in truncation of segments is the updating of quota used figures for the quota account against which the truncated segment is charged. This involves some machination in the program truncate_vtoce to locate this quota account.

The truncation of active segments is performed entirely by pc$truncate, there is not as much as an error code in this case. Records are not deposited, but rather, "nulled", by page control, as described in "Truncation" under "Page Control Services" in Section IX of this document.

The basic steps of truncation are:

1. Determine if the segment is active, which involves locking and searching the AST. If not, it cannot become active, (parent directory is locked) so unlock the AST and proceed with step 2 secure in this knowledge. If active, invoke pc$truncate on the segment, unlock the AST, and return, the truncation being complete.

2. Read in the VTOCE file map. This must be done by obtaining the first vtoce-part, containing the current length and the first part of the file map (also the UID: here is the check for connection failure), and using the current length to determine which other vtoce-parts, if any, are needed. Get them if any.

3. Begin the indivisible operation which must be bracketed by calls to get_pvtx$hold_pvtx and get_pvtx$release_pvtx. Replace the real record addresses in the portion of the file map being truncated with null addresses. Save the addresses in the file map so being replaced, for step 5.

4. Write back the VTOCE with a call to vtoc_man$put_vtoce. Write back only those vtoce-parts which were read in.

5. If there were any record addresses collected in step 3, i.e., real truncation was performed, first await the successful completion of the VTOCE writing started by step 4, via a call to vtoc_man$await_vtoce, and second, upon this successful completion, call pc$deposit_list upon the collection of record addresses gathered in step 3, making them available for use in other segments. This step (5) is skipped for deciduous segments, as their addresses belong to the hardcore partition, and are managed differently (See "Address Management Policy" in Section VII).

6. End of critical section bracketed by get_pvtx calls. Find the record quota account to which this segment's pages are charged, by activating its parent (via a call to activate), and passing the ASTE returned by this activation and the incremental quota change to the page control quota cell manager, quotaw, at entry quotaw$cu.

A fine point of the truncate_vtoce function is the special service performed on behalf of priv_delete_vtoce, described later along with other auxiliary segment control services. If the "owner" field of the supplied branch is "777777777776"b3, which cannot be the UID of any directory, then this branch is a dummy branch for an orphan VTOCE being deleted by sweep_pv. This suppresses step 6 above, as the segment's parent may not even exist, let alone be addressable in this process.

The special treatment of demount protection (i.e., not calling get pvtx$release_pvtx or get_pvtx$hold_pvtx) for calls on behalf of delete_vtoce has already been described under the description of that function.

SATISFYING SEGMENT FAULTS

The most important externally visible manifestation of segment control is that part of it which satisfies segment faults for Multics processes. The technique for using a Multics segment, as implemented by the procedures called through hcs_$initiate, and similar, is as follows: it is called "making a segment known":

1. Use the directory portion of the pathname given to make the parent directory of the requested segment known. When this is done, the Multics virtual memory interprets hardware references to the resultant segment number as references to that directory.

2. Search this directory for the branch that has the entry name supplied to hcs_$initiate in this call.

3. Search the KST (Known Segment Table) of this process, for a segment that has the UID (saved in the KST) the same as the one in the branch found in step 2. If found, the segment is already known; the index of the KST entry is its segment number.

4. If not found in step 3, allocate a new entry in the KST of this process. Put in it the UID of the segment, from the branch found in step 2, and a pointer to that branch. Both are necessary because branches (i.e., segments) can be deleted, or simply moved around by the on-line Salvager. This double-check ensures the binding between branch and segment. Again, the index in the KST of this entry is the segment number.

These operations as described are more properly a part of Address Space Management. The point of restating them here is that they are the preparation in any process for segment control to add the segment to the address space of the process, when that segment number is used in that process. Basically, an attempt to use the segment number gotten in step 3 or 4 causes a segment fault, (directed fault 0, the result of there being "no SDW", i.e., one with sdw.df = "0"b). The segment fault handler (seg_fault, the basis of much of the following discussion) inspects the KST entry in this process specified by the segment number faulted upon (which is in the Appending Unit information in the SCU data stored by the segment fault (see the Multics Processor Manual, Order No. AL39)). The UID therein may be used to find if the requested segment is active; if so, an SDW may be constructed describing the ASTE of the segment. If not active, the segment may be activated from information in the branch of the segment, and then the SDW may be constructed.

Clearly, the construction and use of SDWs, as well as the interrogation of the AST requires all kinds of locking protection, as has been described previously. Thus, this operation of satisfying a segment fault is somewhat more complicated than this. Central to these proceedings is the procedure "activate"; before we describe activation, we first describe the functional interface and purpose of the procedure "activate".

Significance of "activate"

The procedure "activate" is called with a pointer to a directory branch, and returns an ASTE pointer for the segment whose branch was supplied, and a status code. This statement alone says much about what this procedure does; it is the contract of "activate" to make a segment active if it is not, and in either case, return the ASTE (via a pointer) of the segment. Since a decision about whether or not a given segment is active is not even meaningful unless the deciding process has the AST locked, "activate" returns to its caller with the AST locked. It had to lock the AST to find out whether the segment was active in the first place, and once it was active, the usefulness of its activity is limited to operations protected by the AST lock.

The procedure "activate" is given a branch pointer. In general, branch pointers are not valid unless the process using them has the containing directory locked. (The branch pointers in the KST are an exception to this generalization: the UID in the KST entry allows them to be dynamically revalidated every time they are used.) Thus, activate is called, and returns with, the parent directory of the supplied branch locked to the calling process. This fact makes the parent directory lock of a segment implicitly a protection against simultaneous activation; "activate" does not unlock the parent directory at any time.

The operation of the procedure "activate" is thus to obtain information from the branch given (such as the UID),(1) lock the AST, search it for that UID, and return the found ASTE pointer if found, with the AST still locked. If not found, activate proceeds to activate the segment as described under "Activation" below.

SEGMENT FAULT HANDLER

Having set up the necessary framework for understanding of the segment fault handler, seg_fault, we proceed to describe the action taken in response to a segment fault.

The segment fault handler, seg_fault, is invoked by the module "fim" (fault interceptor module, see the Multics Process and Processor Control PLM, Order No. AN60) in response to a directed fault zero. As the segment fault handler returns a zero (successful) or nonzero (error) status code to fim, so does fim restore the machine conditions for that fault (so that the interrupted Control Unit cycle may be retried (see the Multics Processor Manual)) or cause the condition "seg_fault_error" to be signalled at the point at which the fault occurred.

The basic steps of the segment fault handler are as follows:

1. Obtain the segment number faulted upon from the machine conditions at the time of the fault, passed by fim as a parameter. If this is in the range of valid stack segment numbers, and pds$stacks for that number is null, call makestack.

2. Locate the KST entry for the segment (call get_kstep). If this is the root being faulted on, obtain its ASTE pointer (the root is always active: aste.ehs = "1"b, and thus the ASTE need not be locked to use this pointer) skip steps 3 to 6, lock the AST, and proceed directly with step 7.

3. Obtain a valid pointer to the branch of the segment. The procedure sum$getbranch_root_my (see the Multics Address and Name Space Management PLM) is used to do this; it makes the necessary validation checks as described previously, and returns with the parent directory locked, ensuring the validity of this pointer (as well as the existence of the segment and a protection against another process trying to simultaneously activate this segment) for as long as this process leaves that directory locked to it.

4. Obtain access, ring-brackets, entry-bound, and other directory-resident information about the segment from the branch. The procedure update_kste_access is used to obtain the access mode that will be put in the SDW to be constructed. It manages a copy of the access mode kept in the access field of the SDW, and checks whether or not this information is obsolete by comparing date-time-branch modified in the branch given with a copy saved in the KST entry of the segment. If the branch is ahead of the KST, directory control must be called to recompute the access. Recall that this process has this directory locked; no process is now changing the ACL of the segment. See below.

(1) Note that the AST must not be locked to touch directories: see "Locking Conventions", thus it is part of activate's calling rule that the AST is not locked to the calling process at time of call.

5. Check that the logical volume on which the segment resides is either public or private and mounted to this user. Check that it is mounted at all. logical_volume_manager$lvtep and private_logical_volume$lvx provide these services. (See Section XIV of this document.)

6. Call "activate" to obtain an ASTE pointer for this segment, and lock the AST to this process in so doing. As stated, this causes the segment to be activated if not active: other segments may be deactivated in the course of so doing.

7. The AST is now locked to this process, and we inspect the ASTE for the segment being faulted upon. If the referencing address is greater than the maximum length in the ASTE, cause the segment fault handler to return to fim (after appropriate unlockings, of course), so that an error can be signalled. If pack overflow has been observed on this segment (see "Segment Moving" below), invoke the segment mover, and return to fim with the status code returned by the segment mover.

8. Construct a trailer entry in the system trailer segment describing this process' connection to this segment. The fact that we are now committed to constructing and using an SDW means that we must make a trailer entry. See "Trailers and Setfaults" earlier.

9. Compute the new encacheability state of the segment based upon the current encacheability state (see "Encacheability Control" earlier) and the access mode of the SDW being constructed. Directories are generically unencacheable.

10. Build an SDW out of a page table address derived from the ASTE pointer gotten in step 6 (or 2 for the root); mode, ring-brackets and entry-bound derived from the information gotten in step 4 (zero ring brackets, read-write access for any directory); and the encacheability derived in step 9. Install this SDW in the descriptor segment, making it liable to revocation (see "Trailers and Setfaults" earlier) when the AST is unlocked. The process is now said to be "connected" to the segment.

11. Assuming that the operation has progressed this far, unlock the AST (subjecting the SDW to setfaults and the segment to deactivation) and the parent directory (allowing access change, reactivation, or deletion of the segment). Return "no error" to fim.

Some notes on segment fault handling:

The segment fault handler uses the SDW in the descriptor segment as an information repository even at times when the SDW is not valid. These fields (address, ring-brackets, and access entry-bound) are used to avoid recomputation when the reason that the SDW was revoked did not involve changing these quantities. For instance, if a segment is activated and deactivated several times, revoking and re-creating SDWs in many processes, no access or ring-bracket fields need to be changed if no set-acl or set-ring-bracket operations have been performed on the segment. Similarly, if SDWs were revoked because of a set-acl, set-ring-brackets or similar operation, the address in the SDW need not be invalid (or the trailer cut; see "Trailers and Setfaults" above) if the ASTE is not being freed.

Any time that access, ring-brackets, entry-bound, or maximum length (segment bound) of a segment are changed, directory control calls the procedure change_dtem to advance the "date-time-entry-modified" (entry.dtem field of the directory branch). Saving old values and comparing to new values of this pseudoclock can thus be used to see if an older computation of any of these attributes has since been invalidated. This technique is used, as described in step 4 above, to avoid expensive access recalculation in the case of SDW revocation as a result of deactivation. Similarly, the nonzero quality of the SDW field sdw.add is used to avoid freeing and re-creating trailers in the case of access change on an active segment. The procedure setfaults follows these conventions when revoking SDWs, being careful not to destroy these fields of the SDW.

The global transparency attributes (so-called page control switches) aste.gtpd, aste.dnzp, aste.gtus, aste.gtms, (See the ASTE breakdown earlier) are computed from the old values and KST flags each time an SDW is added by the segment fault handler. Thus, segments have these attributes in their ASTE only if the only process that is connected to the segment requests these attributes.

The special case of segment faults on the stack segments of processes is part of the scheme wherein stacks are automatically initialized to the necessary contents for processes to run in the ring of that stack. These references are noticed by the segment fault handler, which does nothing else except call the procedure "makestack", if this has not yet been done for that ring (pds$stacks is an array of per-ring pointers, whose null or nonnull content indicate this). This procedure creates a stack segment, and in initializing it, takes a "recursive" segment fault the first time it touches it. However, it will have changed pds$stacks for that ring to be nonnull by that time, so that segment fault will not be one corresponding to this special case.

A critical aspect of segment fault handling is that any process can "invoke" the segment fault handler (by taking a segment fault) any time it touches any nonhardcore segment _or directory_. Since such segments can be deactivated at any time that the AST is not locked, any reference to a nonhardcore segment (such as user-supplied arguments) or directories is subject to taking a segment fault at that point. Since segment faults cause directories _and the AST_ to be locked, _any process touching user segments or directories can lock directories and the AST as simply a result of such reference_. One implication of this statement is that a process that has a directory locked may not touch any directory or user segment unless it has the following property: A segment fault at that instant would result in locking only such directories that would not cause the process (given that it has this directory locked) to violate the locking hierarchy. One implication of that fact is that _every reference to a locked directory_ is subject to such a segment fault; since a segment fault upon any directory (or segment) will cause locking of its parent, and a directory's parent's lock is higher in the hierarchy than its own (for this _very reason_) directories may be referenced without causing deadly embraces in the case where a process has a single directory (explicitly) locked.

Another consequence of this implementation is that a directory may be referenced with the AST locked to a process _if and only if_ that directory can be established as being active at the time that the AST was locked (for with the AST locked, it, and consequently its parents, cannot be deactivated). Multics does not now make use of this feature. However, the contrapositive of this statement asserts that in general _no directory may be touched_ with the AST locked, for lest it be shown to be active at the time the AST was locked, the resulting segment fault would cause a "mylock" on the AST (which crashes the system), as well as an attempt to lock the (higher) lock of the parent of the directory being faulted upon.

ACTIVATION

The most important step in segment fault handling, the connection of processes to segments, is the activation of the segment faulted upon, in the case where it is not active at the time the segment fault handler locks the AST. The code for activation of segments is in the procedure "activate", whose interface and significance have already been described.

Activation is that action taken by activate when it finds that the segment whose branch was passed in is found, under the AST lock, not to be active.

These are the basic steps of activation:

1. Unlock the AST, having found the segment not active. Since the parent directory is locked, and the segment was found not active, no other process can be attempting to activate it.

2. Get as much of the VTOCE as is necessary to obtain the entire file map. Read the first vtoce-part to determine this; also check the UID of this VTOCE against that in the branch to determine if a connection failure exists; return an error if so.

3. It will be necessary to ensure that the parent of this segment is active (of course, under the AST lock), due to the requirement that all active segments other than the root have active parents. Once we have threaded this segments ASTE into the inferiors list of the parent, it will stay this way. But we must get it this way. This is done by locking the AST, and checking the SDW for this segment to see it has not been revoked (since the AST is locked to this process, it now cannot be). If it has not been revoked, the SDW may be used to find the parent's ASTE (remember that SDWs contain page table pointers, and the page table is in the ASTE). If it has been revoked, unlock the AST, touch the parent, relock the AST and retry this until it is found active under the AST lock. Although a more complex approach that does not involve nondeterministic retry is possible, this action is no more nondeterministic than a process trying to satisfy a page fault.

4. Obtain a new free ASTE for the segment being activated via a call on the AST replacement algorithm in procedure get_aste (see "AST Replacement Algorithm" earlier). This may involve deactivating some other segment (Hopefully not the parent obtained in step 3 -- see below).

5. Thread the ASTE gotten in step 4 into the inferior list of the parent ASTE found in step 3. Fill in the ASTE with all of the VTOCE "activation information" (See the discussion of the VTOCE structure earlier), and initialize cumulated flags (aste.dnzp, aste.gtus, encacheability, etc., see the last section) to default values.

6. Invoke page control (pc$fill_page_table), passing it the VTOCE file map, to initialize the page table and other page control information. Since we are activating this segment, and the parent directory is locked, no one is trying to use this segment, or even knows it is active or being activated, other than this process.

7. Place the UID in the ASTE (see below) and hash it into the AST hash table.

8. Return, with the AST locked, the AST entry (as a pointer) from step 4.

Some subtleties of activation:

The nondeterministic looping and unlocking to obtain the parent ASTE must be done before the obtaining of the new ASTE in step 4. Otherwise, the new ASTE would be in a peculiar inconsistent state during these unlockings. Thus, we determine the parent ASTE before getting the new ASTE. However, there is a distinct danger that the AST replacement algorithm might choose the very ASTE of the parent as the segment to deactivate to provide the new ASTE. Not only would this invalidate the saved pointer to the parent ASTE, but would cause the new ASTE to be threaded as its own parent, causing infinite looping at page control quota management time. Thus, the bit aste.ehs (entry hold switch) is saved, and temporarily set on, and restored, in the parent's ASTE, to prevent the parent from being deactivated by the AST replacement algorithm. The same is true during a boundsfault (see "Boundsfaults" later on).

The UID is the last item placed in an AST entry. This is so that if the system should crash while filling in the AST entry, emergency shutdown could use the fact that the UID is zero as a cue to avoid invoking a VTOCE update on an inconsistent, invalid ASTE. Normally, shutdown (emergency and regular) causes VTOCE updates on all active hierarchy segments. Since the AST hash table manager (search_ast) relies on aste.uid, it cannot be called until step 7 has filled in this field.

DEACTIVATION

Deactivation is the removal of a segment from the AST, the revocation of its "active status". Deactivation is a simple mechanism that is invoked on behalf of the AST replacement algorithm, to free an ASTE to make room for a new one, deletion of segments (see "Deleting Segments", above) to relinquish their AST resources, and volume demounting, to take the segment out of use and update its VTOCE and file map to make the disk being demounted accurate (see Section XIV).

Deactivation, performed by the procedure "deactivate", is composed of the following steps:

1. Check for segments which may not be deactivated, (such as those with the flag aste.ehs on, those with no parent (hardcore) or those with active inferiors). The demand deactivator (see "Demand Deactivation" in this section) can cause this to occur.

2. The AST is locked as a precondition of deactivation. Totally cut the trailer, revoking all SDWs for this segment (setfaults). No process can now use the segment until the AST, at least, is unlocked.

3. Call page control (pc$cleanup) to remove all pages of the segment from the bulk store subsystem or main memory, writing all modified pages to disk (see "Services of Page Control" in Section IX). This resurrects all assigned addresses and finds all zero pages, nulling their addresses (see "Address Management Policy" in Section VII).

4. Update the VTOCE from the now quiescent ASTE, putting final values of file map and all activation information in the VTOCE (see "VTOCE Updating" below).

5. Thread the entry out of inferior lists, decrement parent's inferior count, hash it out of the AST hash table.

6. Make the ASTE free. The put_aste procedure is called to do this: it clears all fields, reinitializes the page table to debugging values, and places the entry at the head of the appropriate used list.

VTOCE UPDATING

VTOCE updating is not strictly a service of segment control or an artifact of its implementation; it is a necessity of the data organization and function of Multics segmentation.

VTOCE updating consists of observing the activation attributes and file map of an active segment, and making the activation attributes and file map in the VTOCE of that segment reflect any changes that have occurred since the VTOCE was last updated, or the segment activated. VTOCE updating is performed routinely every time a segment is deactivated (see "Deactivation" earlier), and when the system is shut down (all VTOCEs of active segments are updated, for both emergency and regular shutdown). VTOCE updating is also invoked periodically by the AST trickle in get_aste (see the earlier discussion "AST Trickle") as necessary, and at certain times in segment moving.

VTOCE updating is performed by the procedure update_vtoce, upon an AST entry (hence the AST is always locked when this activity is performed). In the case of trickle-initiated updates, the information updated may become invalid while it is being updated, but yet, it is a snapshot of some valid state of the segment at some time. The trickle update is a hedge against a fatal crash. Should a fatal crash occur, the pages of the segment that appear in the next bootload, and the state of the segment as a whole, will be that state reflected the last time the VTOCE was updated. Thus the trickle causes periodic and regular update (except under times of very light load) of segments that stay active a long time, and thus, do not enjoy the VTOCE update performed at deactivation. VTOCE updating manifests a critical facet of the system address management policy (see "Address Management Policy, Section VII). Record addresses reported to a VTOCE must be guaranteed to have data from the segment owning the VTOCE, lest the system crash and "uninitialized" pages containing other people's data appear. Furthermore, no record address may ever be freed (added to the free pool of record addresses) unless it is guaranteed that it is not in the VTOCE from which it was culled (See the discussion of "Segment Truncation" earlier in this section).

The steps of VTOCE updating are few and simple.

1. Obtain, from the VTOC manager, as many vtoce-parts as will be necessary to reconstruct the new image of those vtoce-parts that will be changed (see below). For most segments, this is none at all, as the first vtoce-part is usually constructable entirely from the ASTE. (See below).

2. Call page control (pc$get_file_map) to put the latest file map (record addresses and null addresses) in the copy of the VTOCE being prepared. Also, get the latest activation information from a copy ASTE handed out by pc$get_file_map, and put this information in the copy of the VTOCE being prepared. pc$get_file_map also returns a list of record addresses that must be deposited <u>once the VTOCE has been successfully written</u>.

3. Compute and update time-record products if this is the VTOCE of a directory with a quota account.

4. Call the VTOC manager to write out the new copy of the VTOCE, actually initiating its update onto disk.

5. If step 3 returned any record addresses to be deposited, <u>first</u> call vtoc_man$await_vtoce to await the <u>successful</u> completion of the I/O started in step 3, and <u>second</u>, pendent this successful completion, call pc$deposit_list to free these addresses. Again, see the earlier discussion "Segment Truncation".

6. Turn off aste.fmchanged1 if aste.fmchanged was on in the copy of the ASTE returned in step 2 (see below).

It is quite difficult to determine which vtoce-parts have to be read by step 1. If step 3 must be executed, the current time-record product must be obtained, and thus, the first vtoce-part must be read. Otherwise, the first vtoce-part can be written with information wholly derived from the ASTE, and thus need not be read. The second vtoce-part need never be read; either it will be filled with some record addresses and some null addresses as obtained from the file map in step 2, or it will describe a region beyond the current length of the segment when updated, and thus be invalid, and hence not written. If parts of the file map residing in the third vtoce-part must be updated, this vtoce-part must be read, as the permanent information residing there cannot be reconstructed from the ASTE. We cannot know whether or not the third part of the file map will have to be written until step 2 is done. Thus, we make a guess based upon the current length of the segment at the time that step 1 is executed. If, upon getting the current length, it turns out that the segment has shrunk between steps 1 and 2, then the read was unnecessary, and nothing is lost. If, however, we do not read it, and the segment grows, we then read it after we have gotten the snapshot in step 2.

The entry point pc$get_file_map turns off the "file map changed" bit in the ASTE, aste.fmchanged. The semantics of this bit are that the file map has been changed since the last pc$get_file_map. When segment control receives that ASTE, with this bit on, and its file map, it is obliged to update the VTOCE. Should the system crash, however, before this is done, but after page control has turned off the bit aste.fmchanged, the VTOCE update performed at emergency shutdown time will not find the bit on, and thus not know to update the file map in the VTOCE. Therefore, page control turns on the bit aste.fmchanged1 when it turns off aste.fmchanged; update_vtoce turns this off once it has updated the VTOCE. Should ESD find this bit on in any ASTE (see the procedure demount_pv), ESD will take its presence as an indication that this has occurred, and reinstate aste.fmchanged.

A file map, as reportable to a VTOCE, has changed only when addresses are resurrected following successful writes (See "Address Management Policy") or when pages have become zero. However, page control turns on fmchanged when records are allocated to a segment (at new-page fault time) even though they may not be reportable to the VTOCE. A VTOCE, when updated in this state, will have vtoce.records reflecting the real number of records used by the segment (including the new ones) but the file map will not have these new addresses. Should the system crash fatally (no ESD) before such a segment is again updated, or deactivated, the Physical Volume Salvager will notice that records-used is inconsistent with the file map, implying that pages have been lost in this way.

DESCRIPTOR SEGMENT MANAGEMENT

Segment control provides the service of removing descriptors (SDWs) from descriptor segments, in addition to that of creating and installing them (segment fault handling). Often, this service is performed on behalf of segment control itself, such as during the deactivation of a segment, when all SDWs must be revoked. (See the earlier "Segment Fault Handling", including the "Deactivation" discussion therein). Although segment control, via the segment fault handling mechanism, is the only agency in the system that constructs SDWs for hierarchy segments (other than deciduous SDWs and PDS/KST SDWs), several other system functions require revocation or total removal of SDWs. All of these functions are implemented in the procedure "setfaults". The basis of the revocation and trailer mechanism has already been described in the "Overview and Concepts" section (see "Trailers and Setfaults").

All procedures in directory control that change access attributes, such as ACLs (access control lists) or access class must revoke all SDWs for the segment whose attributes are being changed, if that segment is active. This is so that the segment fault handler will find that date-time-entry-modified has changed, recompute the attributes, and give the process a new SDW. Changing maximum length or entry bound causes this same behavior.

The entry setfaults$if_active is called with the UID of the segment to perform such functions. Internal to this procedure, it locks the AST, hashes in this UID to find if the segment is active, performs the setfaults if so, and unlocks the AST.

Another service of the setfaults routine is to remove the SDW for a segment in a given process when that process terminates the segment. This is done because the process no longer wishes the segment to be addressable; it must be removed from the process' address space, because the segment number will be reused (the KST entry has been freed). It is necessary to invoke segment control to remove this SDW because deleting the SDW implies removing the trailer entry in the system trailer segment describing it (which must, incidentally, be done under the protection of the AST lock, which protects the trailer segment). Were this not done, a setfaults on the first segment would randomly destroy the SDW for the next segment that that process had used with that segment number. This entry to setfaults, setfaults$disconnect, supplied with a segment number, also clears the associative memory of the running processor, to remove this SDW from it should it be there. Of course, it is possible that the segment might not be active at the time a process terminates it; in this case, there is no SDW to revoke, but the access information kept there is cleared out. This service is also invoked at the time a process detaches itself from a private logical volume, to make initiated segments on it inaccessible. (See Section XIV.)

Segment control must also be invoked to destroy descriptor segments of processes being destroyed. Each SDW in such a descriptor segment which is for a segment still active at the time of this destruction, has a trailer entry for the process being destroyed, which must be deleted from the trailer list for that segment. The entry setfaults$deltrailer is called on each such SDW, by the process-destruction primitive deactivate_segs (See "PDS and KST Management" later on). Since this is done en masse for all segments in the descriptor segment of the process being destroyed, deactivate_segs locks the AST and calls setfaults$deltrailer for each SDW with a nonzero "sdw.add" field. If a trailer entry is not found at this time, the message "setfaults: missing trailer" appears and a system crash results.

A special kind of setfaults, setfaults$cache is used by the encacheability control algorithm (see "Encacheability Control" in "Concepts and Overview") to revoke all SDW encacheability control bits.

All versions of setfaults other than setfaults$disconnect and setfaults$deltrailer clear the associative memories of the system to force the changed SDWs to be noticed by the system processors. All setfaults other than system-wide setfaults (other than setfaults$cache, setfaults$deltrailer and setfaults$disconnect) also reset the encacheability state of the segment, as no SDWs then describe it. (This action is inhibited by aste.inhibit_cache for IOI buffer segments and the like: see "Encacheability Control".)

BOUNDSFAULT HANDLING

A boundsfault is the occurrence of an attempted reference to an address beyond the current length of a segment as defined by the SDW bounds field (not the current number of records, etc.) If the maximum length of the segment is equal to or smaller than the current page table size allocated for this segment, then this situation is simply an error and is signalled at the point of the faulting reference. If, however, the reference is within the maximum length of the segment, but beyond the current page table size, then segment control must allocate a new page table, and thus a new ASTE for this segment, being in a larger pool. Therefore, a boundsfault (nonsignalled case) involves getting a new ASTE and freeing an old one, and thus shares some of the flavor of both an activation and a deactivation.

The boundsfault handler is the procedure "boundfault". Like the Segment Fault Handler, it is invoked from the fault interceptor, fim, and causes a machine condition restart or signal depending upon the status code returned to fim. Boundsfaults are technically a sub-case of access violation, detected by the 68/80 processor Appending Unit during the SDW appending cycle (see the processor manual).

The basic steps of a boundsfault are these:

1. From the segment number in the machine conditions, find the branch for the segment, locking its parent directory when so doing (a call to sum$getbranch_root_my, just like in the segment fault handler).

2. Lock the AST, so that the old ASTE can be found. If the segment turns out to have been deactivated by the time we lock the AST, it is just as well, as restarting the machine conditions will reactivate it.

3. Find the old ASTE via the SDW in this process (get_ptrs_$given_segno). See step 2 for the notfound case. Get the maximum length from it (aste.msl). If attempted reference is beyond this, unlock the AST and the directory and cause the boundsfault handler to return an error, causing "out_of_bounds" to be signalled.

4. Setfaults the old ASTE. Again, the AST is locked to us, as is necessary to perform this class of setfaults. This inhibits all processes from referencing the segment via the old ASTE.

5. Obtain a new ASTE from get_aste, via the AST replacement algorithm. Temporarily entry-hold the parent ASTE (which is easy to find in this base, as the son is already active (the boundsfaulted segment, and the parent must thus be active) while so doing, so that the AST replacement algorithm does not accidentally deactivate the parent (See the explanation in the description of the segment fault handler for more light on this problem). The new ASTE is guaranteed to be in a different pool than the old ASTE, for that is why we are taking a boundsfault, and thus cannot be accidentally deactivated in these proceedings.

6. Call page control (pc$move_page_table) to move all ASTE information, including the page table (but not the threads) from the old to the new ASTE, and update all page control data bases necessary to move all of the page table (see "Services of Page Control").

7. Rethread all inferior lists and parent pointers affected. If this is a directory being boundsfaulted on, all of the father pointers of inferior segments' ASTEs will have to be updated to point to the new ASTE. This step is the entire reason for the existence of the inferior list in the AST.

8. Hash out the old ASTE, hash in the new, as the segment is still active, but in a different place in the AST.

9. Deposit (put_aste), or free, the old ASTE.

10. Unlock the AST and the parent directory, and return a zero status code to fim.

Fine points:

The most difficult part of the boundsfault operation is that performed by page control, described in Section IX. This is a consequence of the fact that page tables are permanently associated with AST entries.

Very peculiar machine conditions are stored by the PTW2 prepage append cycle used by EIS decimal instructions. This is a consequence of the design that the computed address for the PTW2 page is developed by the Appending Unit of the processor, and not stored as the Control Unit computed address in the machine conditions. Therefore, both the boundsfault handler and the page fault handler (see Section IX) must be aware of these peculiarities of the machine conditions.

SETTING AND REPORTING ON VTOC ATTRIBUTES

As defined in Section II, VTOC attributes are those properties of a segment that are stored in its VTOCE and/or AST entry, as opposed to its directory branch and associated data structures. Typical VTOC attributes are maximum length, current number or records used, date-time-modified, quota used, quota, time-page product. Typical branch attributes are bit count, author, ACL, names.

Directory control primitives, available both through the gate hcs_ and more privileged gates available to the backup system, have need to obtain this information about segments, and set it. The procedure vtoc_attributes performs all of these functions, deciding when to go to the ASTE, when to go to the VTOCE, and which vtoce-parts to deal with.

There are a multitude of entries to vtoc_attributes, which are all either "set" or "get" entries. All of these entries specify a segment via PVID and VTOC index, usually derived from a branch. These entries also receive a segment UID; this allows the segment to be searched for in the AST, and allows a check for connection failure (as in delete_vtoce and truncate_vtoce; see the introduction to "Segment Control Services"). All of the entries are called with the parent directory of the segment locked, and engage in the locking/nonlocking protocol much as given under "Locking Conventions" in Section II.

The vtoc_attributes procedure is protected by the AST lock when modifying attributes. This is a conservative action.

Some notes:

Whenever vtoc_attributes changes a max-length, SDWs may have to be recalculated. Thus, setfaults$setfaults, the most powerful type, is called to fault all SDWs, causing all SDWs to acquire the new bounds field. Of course, all processes using SDWs for this segment then take segment faults, which wait for the unlocking of the parent directory by the caller of vtoc_attributes.

Whenever vtoc_attributes is asked to report date-time used and date-time modified, it updates these quantities in the AST (in the active case). Date-time-used is <u>always</u> updated (the storage system considers used to mean the same as active, in terms of date-time used), (unless aste.gtus is on, suppressing this), and if aste.fms is on (signifying that page control has noticed modified pages), aste.dtm (the date-time modified in the AST) is updated to the current clock value as well, and aste.fms turned off. This ritual is also performed by pc$get_file_map, which reports date-time-used and date-time-modified along with other activation information to the VTOC updater, update_vtoce. (See "VTOC Updating" earlier).

PDS AND KST MANAGEMENT

Each new Multics process (i.e., other than the initializer) inherits the entire hardcore address space from the initializer with a few exceptions. These exceptions are the descriptor segment, the Known Segment Table (KST) and the Process Data Segment (PDS) of the process, and the segment PRDS (PRocessor Data Segment). This is to say that any reference in a hardcore program, via symbolic link (e.g., "call setfaults$deltrailer" or "if active_hardcore_data$x = 7" etc.) refers to the same segment, when the supervisor is running in any process for all segments with these few exceptions. This is because all of the SDWs for a given segment number in different processes (among the SDWs of the supervisor), are copies of each other, never changed or revoked. However, the segments of the supervisor that belong to a particular process must in fact be different from each other. Thus, a reference to segment 60, resulting from a link to, say, pds$processid, refers to different segments in different processes.

The descriptor segment is not created or destroyed by segment control; it is created by the program "plm", which copies the initializer's descriptor segment (the hardcore region) or deals with prelinked processes as appropriate. It is not managed by segment control at all. The contents and meaning of the descriptor segment are, however managed by segment control, as explained previously under "Descriptor Segment Management" and "Segment Fault Handling".

The Processor Data Segment (PRDS), carried around from process to process by a processor as it switches processes, is similarly not dealt with at all by segment control, as a segment, or as a data base. Its meaning, identity, and purpose are explained in the Multics Reconfiguration PLM, Order No. AN71.

The PDS and KST of a process, however, are segments in the storage system hierarchy, in fact, in the process directory of the process to which they belong. The have VTOCEs, branches, and AST entries at times as any other storage system segments. These segments are created by the hardcore process creation program (act_proc), and deleted by the hardcore process destruction program, using the normal directory control segment creation/deletion primitives, append and delentry. In this respect, these segments are peculiar only insofar as that they are created at a validation level of zero, in the ring-0 supervisor. The process creation primitive fills in the new PDS with all relevant and useful information about the new process, having appended it as a segment to the hierarchy, and initiated it as is usual.

However, the use of a piece of the hierarchy as a piece of the supervisor requires special treatment. Note that all deciduous segments are both part of the hierarchy and part of the supervisor (examples: hcs_, sys_info, active_all_rings_data). They, too, are in directories, have valid pathnames, and are described by SDWs constructed by other-than-segment-fault means. These hardcore SDWs, however, which all processes inherit, were produced by initialization, and are not subject to revocation or destruction in any living process. They have no trailers. Now, since these segments are part of the supervisor, in all processes, they may not be deactivated, nor the SDWs revoked, lest the supervisor take a segment fault while performing some operation, such as processing a page fault or a segment fault, which would make this cumbersome, if not impossible. The segment-fault handling code, and all that it relies on (virtually all of the supervisor, as may be inferred from the previous sections) thus cannot be deactivated, nor have its SDWs revoked. There are no KST entries or branches for such segments. They are supposed to handle segment faults, not be subjected to them.

Thus, those segments that will be used as part of the supervisor in a new process must acquire something of the nature of deciduous segments; having nonrevocable SDWs that describe nondeactivatable AST entries. When a PDS and KST have been readied by process creation for a new process, segment control is invoked to transform these segments into reverse deciduous segments, segments which were created in the hierarchy and become part of the hardcore address space. The procedure activate_segs is responsible for this.

The task of activate_segs is making a PDS and KST nondeactivatable, and returning SDWs for them, describing the ASTEs in which they were nondeactivatably activated. The procedure grab_aste, described below, is used to activate them nondeactivatably. When they have been semi-permanently activated, activate_segs returns their SDWs, with the "encacheable" bit on, as explained under "Encacheability Management". For the PDS, a special operation known as "prewithdrawing" is performed. This means that record addresses are assigned to all pages of the segment, to ensure that this PDS, when used as a ring-0 stack in the new process, never is unable to grow a page or itself because there is no more room on the pack that it was on. The PDS cannot be subject to segment moving, when in use by the new process, for it is the very segment that the segment mover uses as a stack in that process. For the KST, we are content to let the process terminate if this highly unusual event happens. For the PDS, however, the system is not even able to invoke the process-terminating software were the PDS unable to grow, and the system loops and/or crashes.

The prewithdrawing is accomplished as follows:

1. The segment has been forcibly activated, nondeactivatably.

2. The bit aste.dnzp is turned on, indicating that no addresses should ever be reported by page control to update_vtoce, thus all addresses ever assigned to this segment stay there (see "Address Management"). This bit is now updated to the VTOCE and reactivated to the ASTE should this segment be deactivated.

3. The segment is released from being held active (grab_aste$release).

4. Each page is touched. This causes a device address to be assigned to each page.

5. The segment is reforcibly activated. It may have been segment-moved in step 4.

At process destruction time, simply releasing these segments from forced activity (grab_aste$release) reverts them to their normal status.

SEMI-PERMANENT ACTIVATION (GRAB ASTE)

The procedure grab_aste is used by the PDS/KST forcible activator as described above, and the IOM/FNP660 Communications Processor buffer facilities as described below. It has a dual task; given a segment pointer (implying that the segment is known in the calling process, and a length, it must activate the segment into an ASTE capable of containing a segment of at least that length, and while the AST is locked, turn on aste.ehs so that the segment becomes nondeactivatable while the AST is unlocked, and unlock the AST and return the AST entry pointer. Since it ensures that the segment is nondeactivatable, the AST entry pointer is valid even after the AST is unlocked.

The steps for forcibly activating a segment into a given-sized ASTE are as follows. The basic technique is to force the segment to be that size, and then activate it.

1. Locate the branch of the segment, thereby locking the parent directory to this process. This, as in the segment-fault and boundsfault handlers, is done via a call to "sum".

2. Save the word of the segment at the given length. Store something nonzero into it. This may cause a segment fault, and may cause a boundsfault. This is valid, for we do not have the AST locked, but we do have the parent directory locked. The segment fault and boundsfault handlers are both prepared to deal with a "mylock" (this lock is locked to my process, so neither will lock it or unlock it) situation.

3. Invoke "activate", as described under "Segment Fault Handling". This procedure returns with the AST locked, and the segment active, and tells us where.

4. Using the ASTE pointer gotten in step 3, turn on aste.ehs (the entry hold switch). This means that the ASTE pointer is still valid when the AST is unlocked.

5. Unlock the AST. The ASTE pointer gotten in step 3 is still valid, for in step 4, the segment became nondeactivatable.

6. Restore the contents of the word changed in step 2. Remember, the parent directory is still locked.

7. Unlock the parent directory.

8. Perform cache machinations as described below if this is grab_aste$grab_aste_io.

9. Return the AST entry pointer gotten in step 3.

The entry grab_aste$grab_aste_io semi-permanently activates IOM and FNP6600 buffer segments (the FNP bootloading segment, IOI buffer segments). As described under "Encacheability Control", these segments must be made irreversibly nonencacheable before subjected to such use, as the processor cache management policy cannot be cognizant of main memory changes produced by the IOM. Thus, when called at this entry, step 8 sets the cache state to "Non-encacheable, multiple SDWs", and sets aste.inhibit_cache so that a set-acl operation will not change this state. It then calls setfaults$cache to revoke all SDW cache bits, so that this nonencacheability takes effect.

The releasing entries, grab_aste$release and grab_aste$release_io, simply turn off the bit aste.ehs, and in the I/O case, aste.inhibit_cache.

IOI AND FNP6600 BUFFER SEGMENT SPECIAL-CASING

As described immediately above, and under "Encacheability Control", segments to be used as I/O buffer segments by the I/O interfaces or in bootloading the FNP6600 Communications Processor, must receive special treatment by segment control. When actually in use as buffers or bootload segments, AST entry pointers to these segments are saved in I/O Interfaces data bases, and page control performs unusual acts upon these segments which prohibit their deactivation during such use. All of these restrictions boil down to the fact that the segments must be semi-permanently activated, for I/O use, as explained above under "Semi-Permanent Activation". Both MCS and the I/O interfacer deal with grab_aste.

SEGMENT MOVING

Segment moving is the single most involved and esoteric action performed by segment control. A segment move is what happens when an attempt is made to grow a segment, there is no more room on the pack, and the segment is wholesale moved to another physical volume in the logical volume where there is room to grow, transparently. Segment moving may also be invoked on demand, via the highly privileged gate hphcs_, in order to move segments between packs to rebalance them or compress a logical volume (remove a pack from it). These online utility operations are coordinated by the online pack utility, sweep_pv.

The essence of segment moving is that it is basically a creation of one segment and a deletion of an old one, as seen by segment control and page control. However, all of the remainder of the system, particularly directory control and the user ring, must see no change; the new segment must replace the old segment, and its contents, in situ. In this regard, it shares some of the flavor of a boundsfault, where one ASTE for a segment replaces another, wholly and entirely in the AST hierarchy (see "AST Hierarchy" in Section II).

The creation of a new segment to replace an old one involves the creation of a new VTOCE. All of the attributes, permanent and activation attributes, other than the file map, of the new segment must be the same as the old. The new segment must have the same contents and unique ID as the old; thus, it is the same segment, once the segment move is over. The directory branch must be changed to designate the new physical volume and the new VTOC index.

Directories may be moved as well as segments. This complicates matters only insofar as AST hierarchy threads must be reorganized in such cases.

Segment moves are provoked either by a call from the interface vacate_pv (See "Special Services for sweep_pv" later on) or as a result of a condition known as pack overflow (or "out of physical volume, 'OOPV'") detected in the segment fault handler.

Page control, upon trying to grow a page for a segment, notices that there are no more records available on its current volume of residence. This may only happen in response to a page fault (see Section IX). The situation requires actions that cannot be invoked by page control, which may deal only with wired data bases in the environment in which it handles a page fault. Therefore, it sets on the bit aste.pack_ovfl in the ASTE, sets a fault in the page-faulting process' SDW for this segment, and restarts the machine conditions. This causes the process to take a segment fault. The segment fault handler (See "Segment Fault Handler", earlier) finds the ASTE, and notices this bit, and calls the segment mover (segment_mover). Upon return from the segment mover, the segment has either been moved (in which case a zero status code is the result) or not (in which case an error code, probably error_table_$log_vol_full is returned), and the resulting error code is returned to fim to signal or restart the fault. When the segment fault is restarted, another segment fault occurs (the segment mover will have revoked all SDWs for the segment, even though page control revoked the one in this process), and the process reconnects to the "new" segment. When that segment fault is restarted, a page fault occurs and the segment, now on a new volume, grows as intended.

The segment mover is invoked, and returns, with the AST and the parent directory of the segment to be moved locked. It does not unlock this directory. It locks and unlocks the AST many times during the course of the segment move. It is passed the ASTE pointer (ensured valid by the lock) and the branch pointer (which it may not use until the AST is unlocked) by the segment fault handler, describing the "old ASTE".

The most basic outline of the segment-move operation is as follows.

1. Make the old ASTE inaccessible with a "setfaults".

2. Create an ASTE (the "new ASTE") for the new segment. (It cannot be activated, for no-one except segment mover can distinguish it from the old ASTE, which is active.)

3. Call create_vtoce$createv_for_segmove (see "Segment Creation" earlier in this section) to create a new segment, given the branch of the old, on some other, suitable physical volume, to create a "new VTOCE".

4. Copy the contents of the segment as it now stands (the segment is unambiguous; it is designated by the segment number faulted upon in this process, the VTOCE, ASTE, and branch it had before the segment mover was invoked) into the VTOCE-less, branchless, anonymous, segment described by (defined by) this "new ASTE". This segment is on the "new" physical volume. Null pages are not copied, to avoid withdrawing records.

5. Copy all the activation attributes from the old ASTE to the new ASTE, make the new ASTE describe the "new VTOCE" from step 2. Update that VTOCE from the new ASTE. Both ASTEs and both VTOCEs now describe identical segments with identical attributes.

6. Change the directory branch (remember, we have the directory locked) to describe the new VTOCE (i.e., change entry.pvid and entry.vtocx). The old VTOCE is now an impostor, the new one is real. Even a crash at this point would affirm this.

7. Unthread and unhash from the AST the old ASTE, thread in (including AST hierarchy threads) the new ASTE, and hash it in as the valid ASTE for the segment under consideration.

8. At this point, the move is essentially complete. The old VTOCE and the old ASTE describe a segment that is not designated by any branch in the hierarchy: an active orphan, not threaded into any structure in the AST. The new VTOCE, the new ASTE, and the branch are consistent. Truncate the segment described by the old ASTE, releasing its disk, bulk store, and main memory resources (it is inaccessible). Free the old ASTE (call put_aste). Free the old VTOCE (call vtoc_man$free_vtoce).

9. The segment move is complete. Return to the segment fault handler or vacate_pv.

The segment mover uses a vast artillery of complex supervisor programming techniques. It involves many of the mechanisms described already, such as segment/VTOCE creation/updating/truncation/deletion, and VTOCE successful-write awaiting. It protects both old and new physical volumes against demount (see Section XIV) during critical regions. There is not much to be gained by a detailed analysis of this little-used and obscure program, when the listing can be read. The outline above indeed explains the basic flow; a few more points will be illuminated, which are critical to the understanding of the basic machination of this operation.

In a situation where a physical volume has experienced pack overflow, it is likely that the logical volume is near full, and all packs or many in the logical volume are near overflow. Thus, if the normal VTOCE creation primitive were invoked on behalf of the segment mover, the volume it chose (See "Segment Creation" earlier) might in fact overflow while step 4 above was being executed. Then the segment mover would recurse. At any rate, the segment mover is prepared for a pack overflow on the new physical volume during step 4, by means of a condition handler for segment_fault error (in this case, an invalid segment number will be the cause of the segment-fault error, although aste.pack_ovfl will be on in the new ASTE). However, even given this, the second choice of a physical volume, should this target pack overflow occur, cannot be influenced by the fact that this first overflow occurred. Thus, segment_mover needs and has a way of trying all physical volumes in the logical volume in sequence, walking the logical volume PV chain (See "Segment Creation" earlier) as a coroutine with create_vtoce. This is to say that create_vtoce is called in a loop on each segment move, at a special entry that walks down the chain finding one acceptable physical volume each time, until segment_mover can perform step 4 without an overflow on the "new" physical volume. A variable (corout_pvtx) passed between segment_mover and create_vtoce$createv_for_segmove keeps track of how far down the chain create_vtoce has gone for this segment move. If step 4 fails on every physical volume though acceptable in the logical volume, or there are none (one criterion on acceptability is at least as many records free as the "old segment" had PLUS the new record that started this all), the segment move fails with error_table_$log_vol_full. Needless to say, more arcane machination is performed when step 4 fails in order to relinquish the VTOCE gotten in step 3, recoordinate all of the data bases and retry steps 3 and 4.

The page control entry pc_wired$write_wait is called at several points in the segment mover. The purpose of doing this is to force all pages of zeros in main memory to be noticed by page control, and "nulled" (see "Address Management Policy," Section VII), to shrink the segment to its minimum possible size (number of records). As a matter of fact, if this operation, performed upon the original segment yields ten or more records, the pack is no longer considered to be in an overflow state, and the segment move is abandoned and declared successfully over. This cannot be the case for segments activated by vacate_pv.

The segment mover updates VTOCEs and deposits record addresses several times; all necessary protocols about waiting for successful write completion (calls to vtoc_man$await_vtoce) are followed.

The updating of record quota used of a directory from old to new ASTEs is difficult, as active segments inferior to a directory being segment-moved may be shrinking and growing.

The segment mover makes use of the segment number by which the segment being moved was known in the running process to construct an abs-seg (see Section VII) with which to reference the old segment; the original SDW was removed by a setfaults call in step 1 above. The abs_seg "abs_seg" is used to reference the segment represented by the "new ASTE". A recursive pack overflow on this segment therefore causes an immediate seg_fault_error, as the segment fault handler refuses to deal with hardcore segments. This causes a signal, that is caught by step 4, and avoids getting into the segment mover recursively although page control induced a pack overflow on the ASTE and revoked the SDW for abs_seg in this case the same as a pack overflow not encountered during a segment move.

SPECIAL SERVICES FOR sweep_pv

The online pack maintenance tool sweep_pv (see the Multics Operators' Handbook, Order No. AM81) can be used to perform operations upon VTOCEs directly from a highly privileged process. Among these operations are:

1. Listing the VTOC of a pack, i.e., reporting the pathnames of the segments owning all VTOCEs.

2. The location of all orphan VTOCEs, (see Section II), VTOCEs not described by any branch in the hierarchy.

3. The deletion of such VTOCEs.

4. The rebalancing of packs via demand segment moving.

5. The vacating of packs (moving of all VTOCEs) via demand segment moving.

The fundamental primitive used by sweep_pv is phcs_$get_vtoce. This entry, supplied a PVT index and a VTOC index, calls vtoc_man$get_vtoce to retrieve this VTOCE, and copies it into the caller's buffer. This entry is not, in its current implementation, protected against volume demounting; it is the user responsibility of the sweep_pv command not to demount volumes to which sweep_pv is being applied.

This entry alone is enough for listing of VTOCs and orphan location. The UID pathname in the third vtoce-part is used to locate a hierarchy branch (develop a pathname). The on-line subroutine vpn_cv_uid_path_$ent performs this UID path (with segment UID) to pathname conversion. This subroutine recursively scans directories by picking them out from ring zero. If this subroutine indicates that either the segment UID in the VTOCE or some UID in the UID path is not the UID of a segment/directory in the directory it claims, an orphan is indicated.

The highly privileged gate hphcs_$delete_vtoce is used to delete orphans. It will delete any VTOCE, be it an orphan or not. The exact description of the act of deleting a VTOCE of a nonorphan is that a (forward) connection failure is caused. There are no tools to cause connection failures in this manner. This gate calls the program priv_delete_vtoce to do the work. This program locks the parent directory; the UID of the parent directory is determined from the VTOCE to be deleted (which is checked, by the way, against a UID supplied by the caller). Note that all that is needed to lock a directory is its UID, notably not a pointer to that directory. The AST is locked and checked to make sure that the segment is not active; if active, it is surely no orphan, and ordinary means (such as the "delete" command (see the Multics Programmers' Manual Commands and Active Functions, Order No. AG92)) may be used to delete it. The operation is aborted in this case, with error_table_$illegal_deactivation as an outcome. The AST is then unlocked; a dummy branch is then created for the segment in the stack frame of priv_delete_vtoce. It has the field entry.owner equal to "777777777776"b3, which will suppress quota movement by truncate_vtoce. The normal program delete_vtoce (see "Segment Deletion" and "Segment Truncation" earlier) is then called, being passed the dummy branch, which has been filled with the physical volume ID and the VTOC index in that volume. The directory is unlocked, and the error code of the delete_vtoce command returned.

The motivation for deleting orphans is not only that the VTOCE is unusable; the VTOCE designates pages in its file map that are unusable. The physical volume salvager does not know that such a VTOCE is an orphan, therefore, its pages are not recovered until the VTOCE is deleted by this means.

The priv_delete_vtoce primitive has a deep dread of accidentally deleting something that is active. It has no qualms about deleting some VTOCE whose segment is not active, and causing a connection failure for that segment. If the UID in the third vtoce-part is correct (not damaged in some unspecified way) the locking of the parent directory and AST scan ensure that the segment cannot be active, or it will be found if it is, and the operation aborted. But, should the third vtoce-part be damaged, AND this primitive is invoked (maliciously) on some segment which is active (sweep_pv, of course, will not do this) chaos will result when that segment is deactivated into a VTOCE which some other segment owns (reused VTOCE syndrome). The crash message "vtoc_man: UID = 0 in a free VTOCE" at some later time will be one of the outcomes of such behavior.

The sweep_pv tool may also be used to force segment moves, either for the purpose of vacating a pack or rebalancing a logical volume. Three primitives are provided for this purpose.

1. The entry vacate_pv$vacate_pv, invoked via hphcs_$vacate_pv, which makes a volume unacceptable for segment creation, whether on behalf of the segment mover or normal creation (pvte.vacating is turned on, which is respected by create_vtoce at both entries).

2. The entry vacate_pv$stop_vacate, invoked via hphcs_$stop_vacate_pv, which reverts the state set above.

3. The entries vacate_pv$move_seg_file and vacate_pv$move_seg_set, invoked via hphcs_$pv_move_file and hphcs_$pv_move_seg.

The vacate and vacate-stop entries are used in two ways: sweep_pv turns on vacating (inhibits) volumes being vacated or moved from, and uses this feature as a control to target segment moves in such operations. These features are directly accessible to the privileged user via the tool inhibit_pv. (See the Multics Operators' Handbook, Order No. AM81.)

The sweep_pv tool uses hphcs_$vacate_pv and hphcs_$stop_vacate_pv to inhibit all volumes, in the physical volume chain of the logical volume on which moves are taking place, between the beginning of the chain and the one where it believes is best for the move to be targeted. As explained in "Segment Moving" before, the mover finds the first acceptable volume to target a given segment move. Thus, the "optimizer" internal procedure of sweep_pv uses these "vacating" bits to manipulate and corner the segment mover, to obtain a balanced distribution of segments and pages, particularly in the case where a volume is being vacated. The sweep_pv optimizer is baroque; read the listing for any more detail.

The demand segment move entries, vacate_pv$move_seg_seg and vacate_pv$move_seg_file, are used to force segment moves on a given segment. As explained above, sweep_pv targeted the move by manipulating "vacating" bits; these entries specify no target volume, the source volume is wherever the segment resides. Both these entries operate by locating the branch for the segment, using either directory control or address space management primitives as necessary, making the segment known (irrespective of the caller's access to the segment), calling activate (see "Segment Fault Handler" for a discussion of the significance of calling activate), and invoking the segment mover upon the ASTE and the branch in hand. The entry to the mover is the same as the one used by the segment fault handler: the only difference is that a referencing address of -1 (corresponding to the address page-faulted upon which causes a segment move) tells the mover that there is no referencing address. The segment is made unknown and the directory unlocked upon completion (the segment mover unlocks the AST).

SERVICES ON BEHALF OF THE HIERARCHY SALVAGER

The hierarchy salvager, when operating in other than 'online-salvager' mode, recursively walks the tree of the Multics hierarchy, walking downward to find directory and segment branches, and returning upward to accumulate and verify quota and quota used totals. The hierarchy salvager maintains its own mechanisms for activating and deactivating directories to be scanned; this is basically historical in origin, dating from the times the the hierarchy salvager was a stand-alone subsystem. In order to perform these activations and deactivations, the salvager must utilize the services of the VTOC manager in order to access and update the VTOCEs of the directories being activated. When running in "-check_vtoce" mode, the hierarchy salvager also reads, inspects, checks and updates VTOCEs of all segments.

The procedure "salv_check_map" in the hierarchy salvager is used by it to read VTOCEs, calling the "get_vtoce" entry in the VTOC manager as appropriate. This procedure maintains an array of VTOCE images, with one entry for each level of directory (and the last level, possibly a segment at each instant) being scanned. During the checking of the branch for each segment or directory, performed in salvage_entry, the parameters in the VTOCE are cross-checked and updated. This includes the primary name, UID pathname, and branch relative-pointer in the "permanent information" in the third vtoce-part. (Again, we reiterate that this checking is done for directories *all* the time, and for segments only when the salvager is "checking VTOCEs", i.e., in "check_vtoce" mode). When the salvager comes back *up* the hierarchy, salvage_directory accumulates recursive information for inferior quota and used figures for each directory being salvaged and includes this among the information being checked by salvage_entry in the VTOCE for that directory. At the end of processing each branch, the procedure "salv_truncate" is invoked. This procedure serves principally to write out the (possibly modified) VTOCE by calling the "put_vtoce" entry of the VTOC manager. If invoked at an appropriate entry, salv_truncate also frees all records claimed by the file map of the VTOCE, thus destroying the contents of the segment. When this is done, salvage_entry, which requested this service, usually destroys the branch as well, and salv_truncate frees the VTOCE via a call to vtoc_man$free_vtoce. This is the hierarchy salvager's mechanism for destroying segments, used in such cases as connection failure, totally unrecoverable directories, etc.

As stated before, the hierarchy salvager has its own mechanism for activating and deactivating directories; it must activate directories in order to check their contents for whatever qualities it seeks. It never activates segments.

Since the entire processing of directories is done as part of the branch checking *for that* directory, (this is to say that salvage_entry, the branch processor, calls salvage_directory, the recursive directory processor, during other branch checks), the time during which each directory need be activated is completely contained in the time during which the VTOCE for that directory is in the array described above (salv_data$vtoce), having been read there by salv_check_map, and to be written out by salv_truncate. The procedure salv_activator is used to maintain a set of sixteen ASTEs, associated with the segment numbers for page-table abs-segs salv_abs_seg_00 to salv_abs_seg_15, into which directories are activated and deactivated from the array salv_data$vtoce as the hierarchy salvager goes up and down the hierarchy. This number corresponds to hierarchy depth. The program salv_activator calls the page control entries usually used by the storage system activation and VTOCE update functions, pc$fill_page_table and pc$get_file_map, to fill and find information about these ASTEs. The entry pc$cleanup is also used by salv_activator, as in normal deactivation, to finalize the state of a segment (See "Deactivation" under "Segment Fault Handling", earlier in this section.)

It is possible that a directory grows during salvaging; in this case, pages are withdrawn in the usual manner; the directories being salvaged reside on whatever volume they do, and are so marked in the ASTE set up by salv_activator, via the field aste.pvtx. The growing of pages against directories is noticed at the time salv_activator "deactivates" each directory, for in this case the bit aste.fmchanged is on. The shrinking of directories by the hierarchy salvager, which can also cause this bit to be turned on, is much more common.

DEMAND DEACTIVATION OF SEGMENTS

The ability to deactivate segments on explicit call is provided via the gate phcs_$deactivate. This is available principally as a performance optimization for the hierarchy dumper. The hierarchy dumper activates large numbers of segments while dumping them. Since it knows that it will never use them after dumping them, it can free its AST resources explicitly, making the ASTEs used by these segments immediately available.

The ability to demand-deactivate segments, as this facility is called, is provided by the procedure demand_deactivate. This procedure locks the AST, checks if the segment specified via segment number is active (the validity of the SDW implies that it is), and if so calls "deactivate" to deactivate it (or fail if it is nondeactivatable; see "Deactivation" under "Segment Fault Handling" earlier in this section). The AST is unlocked, and the error code of "deactivate" returned.

The ability to demand-deactivate any segment is conditional upon the ASTE bit, aste.demand_deact_ok. All processes that have connected to the segment must have had a bit in their KSTEs for this segment stating that they wanted it to be activated with this bit on. Thus, if at least one process is connected to the segment that did not want it to be activated with the possibility of demand-deactivate, it may not be deactivated on demand. This is in order to implement the policy of the demand-deactivation facility being solely a performance optimization for single-process use of a segment when that process fully knows its intended usage pattern for the segment.

One view of this policy is that all activators must agree. Since "normal" use of a segment (via the linker or hcs_$initiate) does not permit demand deactivation, most shared segments (library programs, for example) cannot be demand-deactivated.

SERVICES AT DEMOUNT/SHUTDOWN TIME

The basic goals of demounting a physical volume are to make its contents inaccessible and cause all of the pages and VTOCEs on that volume to contain the latest, up-to-date information. The goals of shutdown, emergency and normal, are the same, except that it applies to each physical volume mounted at the time of shutdown. Therefore, shutdown is implemented as a call to demount each physical volume present at the time of shutdown, with the exception that packs are not unloaded.

Demounting is described more fully in Section XIV. The steps of demounting are these, as seen by segment control:

1. Turn on pvte.being_demounted for the volume being demounted, to cause all activation attempts after this point to fail.

2. Deactivate all segments on the volume being demounted.

3. Turn on pvt.being_demounted2 for the volume being demounted, causing all future attempts to start VTOC I/O to fail.

4. Await the completion of all VTOC I/O for the volume; purge the VTOC buffer segment of all vtoce-part buffers containing vtoce-parts of this volume.

5. Clean up the volume, write out the label, etc. (see Section XIV).

The first two steps stop all activations and deactivate all segments: all attempts to activate check the bit pvte.being_demounted under the AST lock, so that any attempt to activate must either be before or after the AST locking of step 2, and thus either have its activation reverted by step 2 or fail by virtue of finding this bit on as the case might be.

The bit pvte.being_demounted2 is checked by the VTOC manager each time the VTOC buffer lock is locked or relocked; this is the signal of demounting that causes VTOCE operations to unitarily succeed or fail (see "General Policies" in Section III).

The steps outlined above are conducted by the procedure demount_pv, described in Section XIV. Step 4 is conducted by vtoc_man$cleanup_pv, in the VTOC manager, also discussed in Section III.

The deactivate loop in demount_pv, which implements step 2, generally calls the procedure "deactivate" to perform these deactivations; however, in the case of a system shutdown, the critical steps of deactivation, performed by pc$cleanup (finalizing segment state) and update_vtoce (the updating of the VTOCE from the ASTE) are performed by explicit calls to these procedures. This is to avoid dealing with possibly bad AST threads in the case of an emergency shutdown: deactivate generally frees the AST entry being deactivated by rethreading it (via a call to put_aste) in its used list.

The program demount_pv tries to optimize by parallel-processing of many volumes, in the case where many are being demounted. Thus, in its scan of the AST for deactivation, it deactivates segments on any volume that is being demounted. Currently, only shutdown makes use of this feature; normal operator-invoked demounting operates fully one volume at a time.

SECTION V

PAGE CONTROL OVERVIEW AND CONCEPTS

Page control is that subsystem of the Multics supervisor that is responsible for the multiplexing of main memory, the bulk store subsystem, and disk storage. A large part of that responsibility is the transferring of pages of segments between all of these media and the management of the page tables of segments. Page control is also responsible for reporting the status and file maps of segments to segment control (see Section IV, "VTOCE Updating"), and the filling of page tables to make segments addressable by the Multics processor.

Page control has traditionally been regarded as extremely complex and esoteric; this attitude derives in part from the fact that it is largely coded in Multics Assembler Language (ALM), and part from the fact that it is highly asynchronous, maintaining the maximum possible degree of concurrency in all I/O operations. While these concurrency policies will be fully explained, it is assumed that the reader has some familiarity with Multics Assembler Language in order to follow the program listings. A basic familiarity with the appending unit operations (segmentation and paging) of the Multics processor will also be assumed.

The discussion of page control is divided into seven sections in this manual:

Section V.	Overview and Concepts, the current section, explaining basic concepts and goals of page control.
Section VI.	Data bases, breaking down the fundamental data objects of page control, the PTW, the CME, the PDME, the PDMAP header, and the free store maps in the PVTE/FSDCT.
Section VII.	The address management policy used by Multics to avoid accidental disclosure of data by virtue of inconsistencies and crashes.
Section VIII.	The fundamental mechanisms and protocols used within page control to support the services provided.
Section IX.	The services provided by page control to Multics, explained in terms of the mechanisms and data bases described in Sections VI, VII, and VIII.
Section X.	Peripheral services of page control.
Section XI.	Quota management.

The goal of Sections V through VIII is to lead up to the descriptions of the page control services in Section IX. However, these cannot be explained in reasonable terms without comprehension of the information in the preceding sections.

BASIC GOALS AND SERVICES OF PAGE CONTROL

The most visible and crucial service of page control is to handle page faults. A page fault is the fault taken by the 68/80 processor when an attempt is made to append through a page table word that indicates its page is not in main memory. In terms of the Multics virtual memory, a page fault occurs when a reference is made to a page of the virtual memory, a page of some segment, that is not in main memory. It is the duty of page control to allocate a page frame (1024-word block) of main memory, initiate the reading-in or creation of that page of the segment into this page frame, cause the faulting process to wait for the completion of that reading, and notify it so that it might retry the control unit cycle (that sub-portion of an instruction that can be retried with no side effect or regression) when that read has completed.

As part of the mechanism of allocating a main-memory page frame, it is usually necessary to evict some page of some (possibly different) segment from main memory, in order to acquire an unused page. Eviction of a page consists of taking whatever action is required to make a process that might reference that page take a page fault and start these proceedings over again for that page. The choice of which page to evict, or replace, is a critical performance-oriented algorithm of the system. The subject of Page Replacement Algorithms (PRAs) is one covered extensively in the literature, and of great interest to those interested in performance. The Multics page replacement algorithm is described fully under "Main Memory Replacement Algorithm" in this section.

The bulk store subsystem is an optional feature of Multics that allows configurations having relatively small main memories to gain some of the performance benefits of having a large main memory. Under Multics, the bulk store is used as an intermediate-level page storage known as the paging device. Since the average access time (time to access and transfer a page) from the bulk store subsystem is on the order of half a millisecond, as opposed to the tens of milliseconds for the average access time for a page on disk, it is advantageous to the system to keep copies of heavily-used pages on the paging device instead of on the disk. The same is true of main memory; it is advantageous to keep the most heavily-used pages in main memory as opposed to anywhere else. The average access time for pages, over the whole system, is the sum of the products of the access time for each device multiplied by the relative probability of accessing that device. Thus, it is to the system's advantage to keep copies of the most heavily-used pages in main memory, the next-most-heavily-used on the paging device, with all others being accessible only from secondary storage (the disk). Hence, an arrangement known in the literature as a multilevel storage hierarchy exists, where three different media of progressively increasing size, increasing access time, and decreasing cost per bit transfer pages around dynamically in order to optimize the system's average access time for a page. The strategies for managing the paging device, i.e., the replacement decisions, are part of the paging-device management strategy known as Page Multilevel (PML) in Multics, described later in this section.

A less visible service of page control is the assignment and deassignment of disk records. A disk record is a page-size block of secondary storage, which does not cross a cylinder boundary, existing on a given physical volume (pack), and described by its record address on that pack, the zero-indexed integer describing its position in the array of records on that pack. Record addresses (i.e., disk records) are assigned to pages of segments the first time a page of a segment is referenced. They are unassigned at the time that VTOC entries are updated, which occurs most often when segments are deactivated (see Section VII, and the glossary). Record addresses may be nulled or live at any time, while in use in page control, describing whether the record on disk contains data from the page of the segment, or the page of the segment is supposed to contain zeros. The motivation behind these strategies, and their implementation, is a very important part of page control, and is described fully in Section VII, "Address Management Policy." This particular issue also interacts strongly with segment control; (see "VTOCE Updating" in Section IV).

In addition to the transferring of pages between the levels of the storage hierarchy (not to be confused with the storage system hierarchy), page control is responsible for the maintenance of active segments. An active segment, as fully described in Section II, is one which has a page-table in main memory. Page control is responsible for maintaining the current length, record usage, quota information, and most important, file maps, of all active segments. The file map is the mapping between pages of a segment and disk records or pages of zeros. Not only does this include dealing with segments activated and maintained by segment control, but includes segments that have neither VTOCEs nor branches, created by initialization, process creation, etc., and various levels of abs-segs (page tables and ASTEs used for addressing secondary storage explicitly) used all over the system. In the usual case, page control is responsible for filling ASTEs and page tables at the time that a segment is activated by segment control (see "VTOCE Updating," in Section IV).

Page control performs a large and complex set of auxiliary services on behalf of the rest of the supervisor. In part, the need for many of these stems from the fact that a process which takes a page fault may lose the processor while waiting for it. Hence, any code that uses a per-processor resource, such as the per-processor stack used at interrupt time, may not take page faults. Furthermore, any code that is executed under the protection of a lock that has been locked by looping until it becomes unlocked may not lose the processor on which it is executing, lest another process try to lock that lock, and loop potentially forever on a one-processor system, or for an indefinite time dependent on the vagaries of the scheduler in a multiprocessor system. Thus, many diverse portions of the supervisor have a need to avoid taking page faults while they run. Code and data bases that are not subject to partial removing from main memory are said to be wired, and the act of making a set of pages wired is known as wiring, the inverse of this is known as unwiring. All of page control is wired, to avoid taking page faults while processing page faults. There is one special case of a page fault being taken during a page-fault, the so-called "recursive FSDCT page fault." This is explained fully in Section VIII. Thus many subsystems of the supervisor call page control to wire their procedures, stacks, linkage sections, and data bases to perform this class of manipulations. Such wiring is called temp wiring. More fully, temp-wiring is the wiring of a segment or part of a segment by reading in its pages and making them nonremovable by the page replacement algorithm, by covenant with page control. For some segments, like wired deciduous segments (see the glossary, e.g., pl1_operators_) this "temp" wiring is for the life of the bootload. Temp-wiring is as opposed to "perm wiring," which is the act of creating an unpaged segment, i.e., one that does not have a page table, is contiguous in main memory, and whose main memory location and extent are directly described by SDWs that describe the segment. Such segments are made only by system initialization.

One of the implications of the fact that page control itself is mostly wired (perm-wired, as a matter of fact), is that the descriptor segment of any process that uses page control must itself be wired, as were this not the case, page control would take a descriptor segment page fault on the descriptor segment it attempted to run on, hanging up the 68/80 processor in a "trouble fault" loop. Furthermore, the per-process data base in which page control stores each process' page-fault machine conditions must be wired as well. This data base is the PDS, or Process Data Segment, of the process. This versatile data base not only contains page control variables, but all process definition variables, a stack for unrestarted user-ring faults, a pathname associative memory, and entire per-process ring-0 stack. (See "PDS and KST Management" in Section IV for details of segment-control special-casing of this segment.) In order to minimize the amount of this segment which must be wired, therefore, as wiring reduces the total main memory resource available to all users, page control and traffic control, restrict themselves to using only variables and data areas in the first page of the PDS of a process. Similarly, all of the SDWs needed by these two subsystems, and the supervisor as a whole, in fact, are in the first page of the descriptor segment. Thus, the first pages of the descriptor segment and the PDS are called the two critical process pages of each process. Since no process can run unless its two critical pages are wired, a number of pages equal to twice the number of processes that can run must be wired at all times. Since this can be a large number of pages, performance constants require only a subset of all processes _eligible_ to run at any time. The traffic controller gives processes _eligibility_ and takes it away depending on scheduling decisions; a process that is eligible cannot run until it is _loaded_. This loading consists of wiring its two critical pages. Similarly, when eligibility is taken away, a process is _unloaded_. The loading of processes is initiated immediately at the time the traffic controller makes them eligible. The service of loading and unloading processes for the traffic controller is an important auxiliary service of page control.

Page control also provides services to dynamic reconfiguration; when a system controller is removed from the Multics configuration, all pages in page frames in that system controller must be evicted. This can even include wired pages, which involves some machination. Single page frames can be deconfigured via the operator "delmain" command (see the _Multics_ _Operators_' _Handbook_, Order No. AM81 and the _Multics_ _Reconfiguration_ _PLM_, Order No. AN71). Page control must evict their contents, and avoid future use of these frames. Similarly, page control must make available main memory frames that become usable as controllers or individual page frames are added back to the configuration.

The Input/Output Multiplexer (IOM) has a feature whereby a limited form of protection may be used, if the I/O requests for a given channel are constrained to a given region of main memory. The IOM, when performing data transfers and control word transfers for that channel, will not only relocate all addresses found therein with respect to a per-channel "Base Register," but check these (relative) addresses against a per-channel "Limit Register." These IOM features allow the Multics I/O Interfacer to allow users to construct IOM control word lists, and perform data transfers directly to and from user segments. This ability implies that these segments, or portions of them, must be placed contiguously in main memory, not only being wired, but not _movable_ for memory reconfiguration. Such pages are called _abs_-_wired_. They may not be moved because the IOM will have absolute addresses of regions in these pages in its internal registers, which are not subject to manipulation by page control. The service of abs-wiring parts of segments, also used by the FNP6600 Communications Processor bootload software is another auxiliary service provided by page control.

Another service of page control is the so-called "post-purging" feature invoked by the traffic controller. When a process loses eligibility, this function is invoked to bias the page replacement algorithm toward claiming pages deemed "intrinsic" to that process.

Page control also manages _record_ (or _page_) quota. Maintained in active segments' ASTEs and nonactive segments' VTOCEs, quota must be checked, and quota-used totals adjusted whenever pages are created or destroyed. This mechanism is solely for storage system hierarchy segments; supervisor segments have no quota checking.

BASIC ORGANIZATION OF PAGE CONTROL

Page control is said to consist of three major _sides_, or invoking environments, and a few lesser ones. All actions and mechanisms in all parts of page control must take into account the actions of all of the "sides." This organization is also somewhat conducive to the understanding of the organization of the actual modules. The three major sides are:

1. The _page_ _fault_ _side_: the software invoked in response to a page fault in a Multics process, and all software invoked by it.

2. The _call_ _side_: entries invoked by segment control, reconfiguration, initialization, I/O management, etc., to perform all services required by them of page control.

3. The *interrupt side*, or *done side*, named after a routine in the module page_fault. This side is called by the storage system device routines (the *disk DIM*, disk_control, and the *bulk store DIM*, bulk_store_control) to notify page control of I/O operations upon pages that have completed. This side is peculiar in that it may be invoked by the storage system DIMs while other parts of page control have called these DIMs.

The minor sides of page control are those entries called by the traffic controller; those which perform the loading, unloading and post-purging services. These entries are fundamentally different from the others in that they run on behalf of the traffic controller as opposed to on behalf of the process executing them; thus very special techniques for waiting on events, which are not used elsewhere in page control, are used.

Page control may also be divided into the divisions "*ALM page control*" and "*PL/I page control*." Rather than simply indicating the language in which the particular modules are coded, this division emphasizes a fundamental division of functional responsibility. ALM page control is the heart of the entire mechanism. It consists of the entire path taken by a process that takes a page fault, other than the disk DIM and those parts of the traffic controller that are invoked. This includes not only the actual page fault handler, but the fundamental internal primitives that organize the reading and writing and eviction of pages, and the implementations of the page and paging device replacement algorithms. It also includes the logic to allocate disk records. The programs in ALM page control are: page, page_fault, pd_util, free_store, device_control, post_purge, page_error, evict_page, and (by some standards) bulk_store_control, which is the bulk store DIM. ALM page control is sometimes called the page control *kernel*.

PL/I page control consists of all of the call-side functions: entries invoked by segment control, including those for mass *deposition* (deallocation) of disk records. It includes the entries called by reconfiguration, initialization, I/O management, and traffic control (other than post-purging, which is in ALM page control). All of the programs in PL/I page control rely upon the fundamental primitives in ALM page control to do actual deeds; most of the logic in PL/I page control consists of determining which things have to be done, and invoking entries in ALM page control to do them. PL/I page control accesses ALM page control exclusively through the transfer-vector "page," which is there to localize this interface. The most important program in PL/I page control is the program "pc", which, among other functions, contains the entry points that implement all of the services provided to segment control. The other programs in PL/I page control are pc_wired, pc_abs, pc_contig, wired_plm, and by some standards, disk_control which is the disk DIM. There is also "quotaw", which handles quota cells of active segments.

Another important distinction between PL/I page control and ALM page control is that ALM page control works on *pages*; the individual entries each manipulate one page. The PL/I page control entries deal with entire segments or regions thereof, calling ALM page control to perform operations on each page. Other than the page-fault handler, ALM page control never gives up the processor, or *waits*; PL/I page control decides on what to wait based upon a series of calls to ALM page control, and if necessary waits. The protocols involved in this waiting, the conventions used, and the manner of its implementation are all described in Section VIII, "Mechanisms."

There are a set of peripheral services provided by an amorphous area of the system, which could be considered part of page control. For instance, the procedure wire_proc, which causes parts of procedures and their linkage sections to be wired, simply by calling pc_wired, and freecore, which so wires itself in order to make main memory frames available for use as they are added to the system, either during initialization or reconfiguration. These will be dealt with in Section X.

PAGE TABLE LOCK

There exists a lock in the SST (System Segment Table) segment, that protects all of the actions of page control, other than the unloading of processes and activation of segments. This lock is called the "Page Table Lock," or the "Global Page Table Lock." A process that has succeeded in locking this lock to itself is said to "hold the page table lock," "have the page table lock locked," or, often, loosely, "to have the page tables locked" (although the implication that this is solely a lock on page tables is incorrect) or even more loosely, "to have the page table locked." This lock lives in the variable sst.ptl, in the SST segment. It is of the class of locks to which a process that has it locked may not give up the processor until it has unlocked it. This precludes taking page faults. Because certain interrupts try to lock the page table lock, or locks which are locked while it is locked, neither may a process take interrupts while it has the page tables locked. No page faults may be taken with the page table lock locked, and segment faults are out of the question. As a matter of fact, any fault other than a connect or timer runout fault taken by a process while it holds the page table lock will cause the system to crash. This is because page control is not coded so as to be interruptable at any point and salvaged or restarted. Such a recoding is a future possibility.

All sides of page control lock the global lock. Other than on the fault side, this is accomplished by looping on it until it becomes unlocked. The fault side has a special protocol with the traffic controller so that a process which, upon taking a page fault, finds the page table lock locked, can _wait_ via the traffic controller wait/notify mechanism for the lock to become unlocked. This mechanism is explained in Section VIII. A process looping on the page table lock, as it is said to be doing when looping waiting for it to unlock, must be masked so that it may not receive interrupts, or else, as soon as it had it locked, it would potentially take an interrupt with the global lock locked.

It is not necessary to have the global lock locked when activating a segment; since the AST is locked, and before the AST was locked, the segment was not active, no process other than the one performing the activation is aware that the segment is active or being activated. Thus, no process can take page faults or request that auxiliary services be performed upon that segment until the activation is complete. Unloading similarly does not require locking the lock, for as will be described, it involves only the turning-off of two bits that would not otherwise be turned off.

OUTLINE OF THE DATA BASES OF PAGE CONTROL

There are six basic data bases with which page control concerns itself. One of these, the AST entry, is a data object, per active segment, in which information about the segment is kept. A detailed breakdown of the AST entry is given in Section II. Most of the fields in the AST entry are used by segment control; many are used by page control. Those fields are so marked in the description in Section II.

The _page table_ of a segment is that hardware-recognized array, pointed to by the SDW of a paged segment, which converts any reference to that segment to either a reference to main memory, or a page fault. The page table of a segment is physically and logically associated with the AST entry. The page table consists of _Page Table Words_, or PTWs. Each PTW describes the status of one 1024-word page of the segment. If the "4,dl" bit is on, (ptw.df), the upper fourteen bits describe the upper fourteen bits of the main memory address where a reference to that page is to be resolved, the low ten bits coming from the computed address of the 68/80 Control Unit for that reference. If ptw.df is off, the processor takes a fault when an attempt is made to use that PTW. There are also two regions (zones) of the PTW (7000,dl and 700,dl) into which the processor stores 1-bits when that PTW is used, or a reference is made via that

PTW which modifies the contents of the main memory frame it describes. These bits (ptw.phu for used, ptw.phm for modified) are used to determine whether evicting a given page will entail writing it out (if ptw.phm is zero, a good copy exists elsewhere, and to control the page replacement algorithm. The processor associative memory is used to help avoid storing these bits each time such a reference is made, the copies of PTWs in the associative memory contain copies of the ptw.phm bits, and the appearance of the PTW in the associative memory is *de facto* evidence that the "used" bit (ptw.phu) need not be updated.

Page control uses the other fields of the PTW, as well as the "address" field at times when the "fault" bit (ptw.df) is *off* (signifying *take* a fault, *no* access) to store control information. In particular, the bulk store or secondary storage address of a page not in main memory is stored in the PTW in this fashion; when in main memory, this information is transferred to other places, namely, the CME (Core Map Entry).

The core map, so-called from the days before MOS technology became prevalent for main memory), is an array of four-word *CMEs*, or core map entries. Each entry describes the status of one page frame of main memory, including all page control information. There is a core map entry for each page frame in the configuration from address zero to the highest address in the configuration, whether or not a physical controller or memory exists that contains the implied page frame, and whether or not this page frame is available for page control's use (for instance, it may be in the middle of a perm-wired segment). Thus, the core map is an array indexed strictly by main memory address. The core map is in the "SST" segment.

The core map entries are kept in a double-threaded circular list; the (SST-relative) pointer sst.usedp describes the "head" of the list. The list is the basis of the implementation of the main memory page replacement algorithm, which is described later in this section. Entries for main memory frames that have I/O going on are threaded out of the list, as are entries that correspond to main memory not used for paging. Entries that correspond to main memory that does not exist, be it deconfigured or simply not present in the configuration, are threaded out with a thread word of "777777777777"b3. The last word of a core map entry is currently not used.

The paging device map resides in the SST as well, in configurations with a paging device, directly after the core map. It consists of four word *paging device map entries*, or *PDMEs*. It, too, is an array, indexed by record that describes paging device record zero; if only some upper portion of the bulk store is in use as a paging device, this pointer points *below* the start of the paging device map, and possibly below the origin of the SST. This is to ensure that this pointer always points to the virtual origin of the array. The entries of the paging device map are similarly kept in a double-threaded circular list, as befits the parallel problem of management of the paging device already alluded to. Those which have been deconfigured, either by operator "delpage" command, or the automatic deconfiguration performed by the interrupt side on detection of bulk store error, are threaded out with a thread word of "777777777777"b3.

The first few records of the bulk store are not used as part of the paging device; rather, the paging device map is written out from main memory to as many of these first few records as need be to contain it, *every second*. This is done as a hedge against fatal (no ESD) crashing. Should the system crash unrecoverably, the next bootload can read the contents of the first few records of the bulk store, and obtain the old paging device map, accurate to within a second. As physical volumes are *accepted* (see Section XII) by that next bootload, pages of segments on that volume are *repatriated* from the old paging device contents as their VTOCEs are processed by the physical volume salvager. A Unique ID and page number are put in each paging device map entry to facilitate repatriation; because of these two quantities, the second inaccuracy of the paging device map need not be a cause for concern. Thus, the paging device map has potentially a cross-bootload longevity. To facilitate interpretation of its contents, the PDMAP (as the paging device map is sometimes called, not to be confused with sst.pdmap, which stands for paging device map array pointer) has a four-word header, the pdmap header, describing the extents and time of initialization (called the PDMAP time) of the paging device map. This PDMAP time is marked in the volume labels of all physical volumes which were part of the configuration during which that PDMAP was used; this is the key to the mechanism (explained fully in Section VIII, under "Post-Crash PD Flush") by which pages are repatriated as volumes are accepted. Because the first record of the bulk store contains the first page of the PDMAP, the first PDME of a PDMAP is not used, but contains the PDMAP header. All PDMEs that describe records similarly used by the PDMAP image other than the first are not used at all, and contain all zeros.

The FSDCT is a data base used by volume management (see Section XIII) to record certain key global parameters of volume management. These all reside in the FSDCT header. The remainder of the FSDCT is divided into regions, one for each configured storage system drive. These regions contain the bit-map of free disk records for the packs mounted on their respective drives. The parameters governing the interpretation of that bit-map are in the physical volume table entry for that drive. The physical volume table entry, or PVTE, is an entry in a wired table, the PVT, which describes all parameters for a given drive and the pack on it, used by the storage system. (The PVT and PVT entry are described fully in Section XIII.) Among these parameters is a relative pointer into the FSDCT of the bit-map for that drive, and its extent, number of records still free, etc. Needless to say, many of these parameters, including the entire contents of the bit map, change as packs are mounted and demounted on that drive. The algorithms used to manage this map and allocate free storage are described in Section VIII, "Mechanisms." Some critical points relating to the assignment and deassignment of addresses are given in Section VII "Address Management Policy."

The letters "FSDCT" stand for "File System Device Configuration Table." In light of the current storage system, this term no longer has any valid connotations relative to its meaning. If anything, the PVT deserves that title; it is strictly historical, for in older versions of the storage system, the single large bit-map describing the entire mounted storage system was kept here. The format of the FSDCT bit-map regions and the relevant variables to free storage allocation are given in the detailed data base breakdowns in Section VI.

The FSDCT is not a wired data base. In a system with many drives, it can grow quite large, and would constitute a substantial drain upon the main memory resources of the system were it all wired. Therefore, it is used subject to vagaries of its own dynamic paging behavior. However, one of the critical usages of this segment is the allocation of disk addresses, which is performed during page-fault handling. Since the page-fault handler may not take page faults, there is an intrinsic difficulty in accessing this segment at that time. A very special and intricate mechanism exists to allow the page fault handler to simulate "recursive" page faults on the FSDCT. This mechanism is explained in Section VIII under the heading "FSDCT Paging." Other programs with a need to reference the FSDCT, such as the activation-time check for unprotected addresses (those illegally marked as "free" in the FSDCT) simply reference the FSDCT like any other paged segment.

Other than the FSDCT and PVT, all of the data bases of page control reside in the segment "sst", with the alternate name "sst_seg." This segment, also known as "the SST", for System Segment Table, is an unpaged (perm-wired) segment, in which all AST entries, with their page tables, the core map, and the paging device map reside. All of the page control data objects describe each other via relative, 18-bit pointers, called "rel-pointers," or "SST-relative pointers." The only exceptions to this rule are main memory and paging device addresses, which are effectively indices into the core map and PDMAP arrays.

The SST also contains a large number of meters, list heads, and array pointers. Much global page control data is stored there.

ZERO PAGES

Multics defines all segments as containing a full segment's worth of binary zeros when created. Rather than allocating a couple of hundred disk records and zero them each time a segment is created, Multics defines a class of record address called a *null address* which says that the page that has that address is supposed to contain zeros. That is to say, if such a page is faulted on, page control creates a page of zeros in main memory. Real disk addresses and paging device addresses are assigned at various times after that, as dictated by the address management policy (see Section VII).

In order to keep this strategy consistent, Multics never stores pages of zeros on disk or on the paging device. Whenever a page is to be written out of main memory, a check is made to see if it contains all zeros. If so, the disk address which the page has is *nulled*, creating a *nulled* or *semikilled* address in the page control data bases. Like a null address, the next attempt to fault on this page causes a page of zeros to be created in main memory. If the page is modified to be nonzero, the address is *resurrected*, (made not nulled), which causes a real read to happen when the page is faulted on.

The terms *null* and *nulled* are not to be confused, although both logically represent pages of zeros, the *null* address relates to no disk record; the *nulled* address represents a disk record, but the contents of the page are zero, not the contents of the disk record. Nulled address appears only in page control, never in VTOCs or other segment control data objects.

This checking for zero pages is suppressed for segments with the "dnzp" (Don't Null Zero Pages) attribute settable via segment control, and always true for supervisor segments. This is used, in general, to enforce the requirements of the address management policies described in Section VII.

Nulled addresses which result from the discoveries of pages being zeros ultimately get returned to the free storage pool for their volume; this is done once it is ensured that the un-nulled address from which it came is no longer in any VTOCE. (See Section IV and Section VII.)

MAIN MEMORY REPLACEMENT ALGORITHM

Of fundamental importance to any algorithm that controls the movement of pages, and of prime interest in the description of any paging system, is the main memory replacement algorithm, known in the literature as the "Page Replacement Algorithm," or PRA. The Multics PRA was one of the first to ever be implemented; the version as it exists today is a direct descendant of Corbatô's original algorithm (see the references at the end of the next section).

Pages are kept in a circular list, the *core used list*, implemented by the double thread of CMEs. A logical pointer is kept to a selected point on the list, this being implemented by the SST-relative pointer sst.usedp. A direction called *forward* or *ahead* is arbitrarily defined as the direction on the list followed by chasing the sst-relative pointers cme.fp.

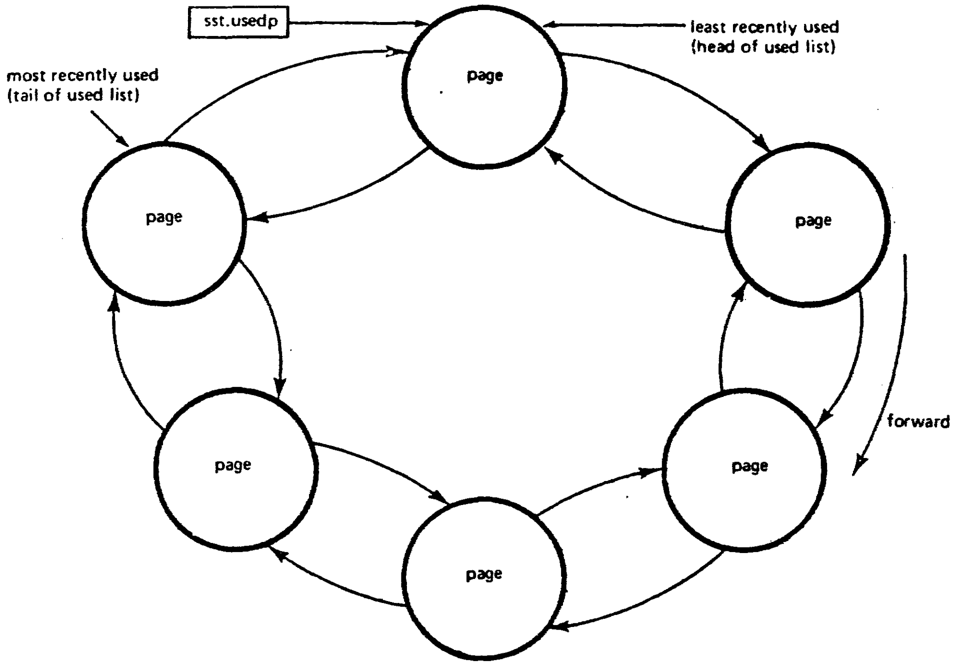


Figure 5-1. The Clock Algorithm

The basis of the algorithm is that the pointer moves forward on demand for page frames. It tries to approximate the "Least Recently Used," or LRU algorithm, where the least recently used page (not page frame) is the one which will be evicted to free its page frame. The page frame right ahead of the pointer (the one pointed to) contains the supposedly least-recently-used page. Going further and further down the list produces pages more and more recently used, until the page right behind the pointer is the most recently used. Since pages are referenced by every instruction that runs, it is impossible to thread them to represent true recency of use. Therefore, we translate "recently used" into "recently noticed as used." When we notice that a page has been used, we turn off the bit ptw.phu, in the PTW for that page, the bit via which the hardware communicates the fact that a page has been used. Thus, this bit being on in a given PTW indicates that the page has been used since this observation was last made.

Therefore, when a demand is made for a frame (via a call to find_core, in page_fault), the page at the head of the used list is inspected to see if it has indeed been used since last inspection. If so, it is now, clearly, the page most "recently noticed as used." Thus, the pointer moves forward, putting this page at the tail of the used list by so doing, in keeping with its newfound status as "most recently noticed as used." The "used" bit is turned off, pending the next inspection, and the next page is considered, until one is found whose used bit is off. Such a page is clearly the one which was seen most recently as used the furthest time in the past. This page is evicted from its main memory frame, and the latter is now free.

The algorithm just described is known in the literature as the "clock" algorithm, as the motion of the pointer around the used list is similar to the motion of a hand of a clock about the face of the clock.

There are several complications to this algorithm. Most important, if a page is found whose used bit is off (this would be evicted, according to the above description) by the scan of the pointer, this eviction would require an I/O operation to perform, namely a write to disk or paging device. If the page has been stored into (modified) since it was brought into that page frame, as the information in its correct form exists only in main memory, and nowhere else. Thus, a modified page whose used bit is off, takes more work to evict than one that is not modified. Specifically, the I/O may take an indefinite time to complete, and the main memory request on hand must be satisfied immediately. Therefore, the pointer skips over pages that are modified, even though they are not used--they will be dealt with shortly. The pointer only stops when a page that is neither modified nor used is found--only this kind can be evicted with no I/O. The page multilevel algorithm also complicates matters some here,there are pages that are neither used nor modified which require I/O to evict, if the page multilevel algorithm wishes to migrate them to the paging device at this time; these pages are called "not-yet-on-paging-device," (ptw.nypd signifies this state). This will be dealt with in the next section.

Therefore, the pointer does not stop until it finds a page that is neither used (since last turning-off of the used bit), modified (since last writing), or not-yet-on-paging-device. Some pages are routinely skipped, such as those that are wired or abs-wired. Pages on which I/O is going on are not even in the list, and are thus not an issue. When such a page is found, it is evicted, and the frame which it had occupied returned to the caller of find_core.

In passing over modified and not-yet-on-paging-device pages, the pointer implicitly left work behind to be done. These pages should be evicted from main memory, but this could not be done on the spot, as the process that needed a page frame could be satisfied immediately with some other frame, not much worse, and could not wait for the indeterminate completion of these writes. Therefore, a procedure called claim_mod_core, in page_fault, exists to do the work which the replacement algorithm decided not to do, in order to satisfy its real-time constraint of producing a usable page-frame on the spot. It runs either at a later time than find_core, or is called by find_core when the latter encounters certain limit situations (see Section VIII). The procedure claim_mod_core maintains a second pointer into the used list, which is sst.wusedp (for "writing" used-pointer). Generally, it is pointing to the same place as the regular "usedp" clock-hand of the find_core command. However, when a demand is made for a page-frame of main memory, find_core advances the "usedp" hand until a freeable, evictable frame is found. Thus, the distance between the "wusedp" hand and the "usedp" is the "cleanup" work that must be processed by claim_mod_core. The procedure claim_mod_core is invoked during page-fault processing at a time to overlap its operation, which may involve substantial computation inside the disk DIM, with the reading-in of the page necessary to satisfy the page fault. Note that this reading could not begin until a page-frame into which to read the page had been found, by find_core. Claim_mod_core processes all page-frames between wusedp and usedp; those that are not used, but modified, have writes started for them, which removes their CMEs from the used list. In order for claim_mod_core to be able to distinguish the used-and-modified ones from the not-used-but-modified ones, find_core avoids turning off the used bits, leaving this for claim_mod_core. Pages "not-yet-on-paging-device" are migrated to the paging device, as appropriate, until wusedp and usedp again coincide. Note that these writes are started while no particular process is waiting for these writes to complete for any reason--when these writes are complete, the interrupt side will place these page frames at the head of the used list, making them excellent candidates for eviction if and only if they have not been used while or after being written.

The interaction of find_core, the replacer, and claim_mod_core, the purifier, may be stated as this: the replacement algorithm claims only pure (unmodified) pages. Those that are found impure, but would have been claimed, are left for the purifier to purify. When the purification is complete, these pages are again candidates for replacement.

There are a large number of call-side actions, such as deactivation and truncation, and some ALM actions, such as the discovery of zeros by the page-writing primitive (write_page in page_fault) that cause page-frames to become explicitly free; these actions all aid the replacement algorithm and simplify its task by putting these page frames at the head of the used list, wherever it currently is, making these frames immediately claimable by find_core.

The successful completion of any read operation places the CME for the frame into which the reading was done at the tail of the used list, as presumedly, the reason that this read occurred is that someone wanted the page, and thus, it is "most recently noticed as used" at the time of the completion of the read.

PAGING DEVICE MANAGEMENT ALGORITHM (PAGE MULTILEVEL)

The management of the paging device, like the management of main memory, involves both a strategy, and a replacement algorithm. In the case of main memory, other than the replacement policy, the strategy is straightforward. Pages are brought in on demand in response to page faults and call-side reads, evicting other pages at the discretion of the replacement algorithm, which also chooses when to write out pages that have been modified.

The use of an intermediate level of storage device as a paging device, however, involves many more complex decisions. The design and history of the decisions, with respect to the Multics Page Multilevel Policy, are given in the paper by Greenberg and Webber cited at the end of this section. The policies are given as they stand.

The paging device is what is technically called a "nonwrite through buffer." This to say, there are copies of pages on it which are different from the copies of the same pages on secondary storage. As a matter of fact, there can be copies of pages on the paging device which have no copy in secondary storage (although there will always be a secondary storage address assigned to such pages). This allows pages to be written from main memory to the paging device without simultaneously writing a copy to secondary storage. (The option to write these pages to secondary storage in this way exists, and is called "double writing," and is controlled by the "DBLW" parameter on the PARM CONFIG card.) If the paging device is operating in double-write mode, or were designed as a "write-through buffer," there would be no damage caused by loss of the paging device during a running system or a crash; pages on secondary storage would always contain the same information, although at a higher cost to access. The fact that modified pages exist (modified with respect to secondary storage, that is), while avoiding the substantial expense of double-writing each page of main memory, but causes a substantial problem of updating secondary storage, both during normal operation and the page repatriation operation of a post-crash bootload.

The paging device replacement algorithm is a critical part of the management policy. It is designed to resemble the "clock" algorithm used in main memory management. However, a unique interaction with the main memory algorithm presents itself; while the eviction of pages from the paging device that are not modified with respect to main memory presents no special problems (page control data bases, namely the PTW, are updated to indicate that the page must be fetched from paging device instead of secondary storage), the eviction of modified pages is difficult. In order to evict modified pages, they must be written back to the disk. This is accomplished by finding a usable page-frame of main memory, reading the page in from the paging device, and writing it out to the disk.

This two-part sequence is called a Read-Write Sequence, or RWS. Were the paging device operated double-writing all the time (write-through buffer), there would be no need for RWSs. However, the fact that the main memory replacement algorithm demands pages of paging device, and the paging device replacement algorithm demands pages of main memory, in order to perform RWSs, presents some difficulty. The solution to this problem, which basically involves "punting" paging device migration when recursion would be created, is explained in Section VIII.

The paging device replacement algorithm maintains a circular used list, as the main memory replacement algorithm does. It is of PDMAP entries (PDMEs), and the head of the list (best candidate for replacement) is designated by the sst-relative pointer sst.pdusedp in the SST. PDMEs that are undergoing RWS are threaded out of the list. Before we discuss how pages are migrated from the paging device, however, it is appropriate to discuss how pages are migrated to the paging device. This has no parallel in main memory management, as pages are "migrated to main memory" as page faults are taken; there is no choice.

Pages are migrated to the paging device as they are evicted from main memory. "Migration" implies that the page does not already have a copy on the paging device. The assumption and design is that the pages that are in main memory, going into it, and going out of it, are the most recently used and thus most likely to be used in the near future, of all of the pages in secondary storage. Therefore, any page just evicted from main memory is more likely to be referenced in the near future than some page less recently evicted from main memory, and it should be allocated a record of paging device, and written to it. Note that this implies writing of pages from main memory that are not different, i.e., not modified, with respect to their copies on disk; these are the so-called "nypd" (not-yet-on-paging-device) pages mentioned in the previous section. The need to do this writing biases find_core against these pages, leaving claim_mod_core to initiate the paging device update. The routine allocate_pd in page_fault is charged with the responsibility of deciding when a page should be migrated to the paging device or have its "nypd" bit turned on to postpone this action.

Some subset of the pages of the paging device are always (nearly always) going to be in main memory. Pages are migrated at main memory eviction time instead of reading time because there is no need to read them back, hence "waste" paging device on them, until they are evicted. It is an assumption of the algorithm that the paging device is substantially larger than main memory; all of the below assumptions fail if this is not true. A paging device smaller than main memory can also cause the paging device replacement algorithm to hang, as will be seen below.

The subset of the paging device, so to speak, which is in main memory, is considered to be the "most recently used" subset. Since the paging device is much larger than main memory, any page found in main memory by the paging device replacement algorithm is promoted to a "recently used," i.e., favored status, similar to that given to pages found with their used-bits on by find_core. No page in main memory is ever evicted from the paging device by find_core, although deactivation or truncation of the containing segment will indeed perform this.

The paging device replacement algorithm is invoked at the beginning of page fault processing, every page fault. It tries to ensure that a small, fixed number (10) of paging device records are always free or in the process of being freed (RWS in progress). Since it does this at the beginning of a page fault, when it is finished, probably some paging device records will have been freed, some already free, some started RWSs, and some finished RWSs from some previous time (made free by the interrupt side). Thus, it is probabilistically very likely that some records will be free during the processing of that page fault (during which claim_mod_core may attempt to migrate pages to the paging device). The replacement algorithm moves down the PD used list, evicting all pages not requiring RWS, and starting RWSs for all pages modified with respect to secondary storage. PD records found to contain pages that are also in main memory are rethreaded in the list so that they acquire the favored "recently seen to be used" status. This action continues until ten records are free or in RWS. There is no problem of obtaining "RWS buffer" pages here, a call being made to find core as each such buffer is needed. Note that find_core will not cause PD records to become allocated in so doing; find_core does not initiate writes. Only claim_mod_core does that.

Thus, by the time claim_mod_core runs, very probably a few records will be available into which to migrate pages, on the paging device. Now it is possible that the page-writing primitive will find that no free records of the paging device are available for migration. Specifically, it looks at the head of the list, checking for the availability of this record. If this record is not available, which will only be the case if no records could be made free by the last run of the replacement algorithm, or there were none when it ran, an action called a PD desperation occurs. The paging device allocator (allocate_pd in page_fault) calls the PD Desparator, (force_get_pd in pd_util) to run down the PD used list up to twenty steps until a claimable PD record (evictable without RWS) is found. If this strategy fails, which it rarely does, the attempt to migrate a page to the paging device, which was an optimization of sorts to begin with, is abandoned, and the system continues normal operation. An RWS cannot be initiated at this time to free up paging device; it would take an indefinite time to complete, and waiting for it in any way would cancel whatever optimization could be gained by migrating the page.

Pages of active segments only (or nonstorage system segments, which are always active) are kept on the paging device. This implies the need to start RWSs at deactivation time, but metering has shown that the number of pages of segments being deactivated which appear on the paging device, and require RWS are few. This scheme avoids the need for repatriation of paging device pages every time a segment is activated. This system was used in earlier versions of Multics, involving the "PD Hash Table" now gone.

One type of event of note in paging device management is the so-called "RWS abort." This occurs when a process takes a page fault on a page that happens to be undergoing RWS. To the process taking the page fault, this is just another page fault. Page control, however, sets a bit in the PDME (pdme.abort), informing the interrupt side not to free the main memory frame and paging device record, but rather to keep both around, and re-establish the residency of the page in both main memory and on the paging device. (Until the occurrence of an RWS abort, pages transiting through main memory in order to perform an RWS are not considered by the rest of page control to be in main memory.)

Papers about the Multics Page Replacement Algorithm:

Corbató, F. J.
"A Paging Experiment with the Multics System," in Ingard, In Honor of P.M. Morse, M.I.T. Press, Cambridge, Mass., (1969), pp. 217-228

Greenberg, B. S.,
"An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory," M.I.T. Project MAC Technical Report TR-127, M.I.T. Dept. of Electrical Engineering, May, 1974

Greenberg, B.S., and Webber, S.H.,
"The Multics Multilevel Paging Hierarchy," in Proceedings of the 1975 IEEE Intercon, Institute of Electrical and Electronic Engineers, N.Y., 1975

SECTION VI

PAGE CONTROL DATA BASES

In this section are discussed, bit by bit and field by field the fundamental data objects manipulated by page control:

1. The Page Table Word (PTW)
2. The Core Map Entry (CME)
3. The PDMAP Entry (PDME)
4. The PDMAP Header (PDMAP Header)
5. The FSDCT bit maps, and relevant PVTE fields.

Also presented is a list of selected fields of the SST data base, with some explanation of their relevancy to page control, and function.

The various data objects are interrelated via 18-bit pointers and radices when in use by page control. Figures 6-1 to 6-5 at the end of the section present the interrelationship graphically for the more important states of those objects.

PAGE CONTROL DEVICE ADDRESS (devadd)

One quantity that crops up in PTWs, CMEs, and PDMEs is the general device address. A device address designates a frame of main memory, a record of paging device, or a record of disk. A device address, or devadd, has two subfields, the address, or record address, as befits which of the above cases is appropriate, and the address type. The bits of the address type are exclusive, i.e., no combinations of more than one bit are valid, and the last bit is reserved. Such devadds appearing in a PTW can designate main memory, a record of paging device, or a record of disk. A devadd appearing in a PDMAP entry must designate a record of disk. A devadd appearing in a core map entry can designate either a record of disk or a record of paging device.

Format of a "main memory address" devadd, valid only in a PTW

```
0                   1 1  2
                    7 8  1
+----------------------+----+
|MMMMMMMMMMMMMM0000    |1000|
|                      |    |
+----------------------+----+
```

top 18 bits of main memory address, "add type," in this case add_type.core.

The main memory address designates a page frame of main memory. It is the upper fourteen bits (MMM...MM) of that address, the remaining ten bits being an address within the page frame. The "1" in bit 18 signifies a main memory address.

Format of a "paging device" devadd, valid in a PTW or CME:

```
0                  1 1  2
0                  7 8  1
|0000000PPPPPPPPPPP|0010|
                    "add_type," here add_type.pd
```

PPP = paging device record number.

The paging device record number specifies a record of paging device. The "1" in bit 20 signifies a paging device address.

Format of a "disk" or "secondary storage" devadd, valid in a CME, PTW, or PDME:

```
0 0                  1 1  2
0 1                  7 8  1
|N|DDDDDDDDDDDDDDDDDD|0100|
                      "add_type," here add_type.disk
```

DDD = Disk record number.

The record number DDDDD is the record address of a disk record, on some physical volume. That physical volume is identified by the PVT index in the AST entry associated with the page table to which the PTW in which this devadd is found belongs. If this devadd is found in a CME or PDME, the volume is identified by the PVT index in the AST entry associated with the page table designated by either of these objects. If this devadd appears in a PDMAP entry in a post-crash PDMAP entry matches the field label.last_pvtx on some physical volume whose field label.pd_time matches the "PDMAP time" of the PDMAP in which this PDMAP entry appears. To that volume this page will be repatriated. (This will be explained in more detail in Section IX.)

The bit "N" above is of prime importance. In this disk "devadd" is the bit "N" (for nulled) being on indicates that although this devadd is assigned to the page in whose data bases this devadd appears, the logical contents of the page are to be considered zeros. Either this page has never been written out or RWSed to that device address, or was truncated, and this page awaits deposition by the VTOCE update function. An address with this bit on is called a nulled or semikilled address; it may never be reported to segment control for a file map, but may only be deposited or resurrected (see Section VII, "Address Management Policy"). These nulled addresses are not to be confused with the null addresses used by segment control in file maps, and below. A disk address that is not nulled is said to be live, meaning it definitely contains the contents of the page to which it is assigned. Nulled addresses appear only on page control.

There exists one more type of devadd, the so-called "null" device address, or "null" address, not to be confused with the "nulled address" explained above. It represents a page of zeros, as does a nulled address, but designates no page of disk. Its format is as follows:

Format of a page control null address; valid only in PTWs:

```
 0                   1 1  2
 0                   7 8  1
|0BBBBBBBBBBBBBBBBBBB|0000|
|                    |    |
                     "add_type," here "null"
```

BBB = debugging code.

The code BBB...B is a code placed in this devadd by the program that generated it, describing how it became null. These codes are described in null_addresses.incl.pl1 and null_addresses.incl.alm, which has some in their "page control representation" as above, and some in their "segment control representation," as below.

Null addresses enter page control from the activation of segments, as well as by other means. Null addresses are also reported to file maps for the VTOCE update function. When in file maps, coming into or out of page control via pc$fill_page_table or pc$get_file_map, page control null addresses are converted (from or to, respectively), the format in which they appear in file maps:

Format of a segment control, or file map null address, never valid in page control, only valid in file maps in VTOCEs:

```
 0                 1
 0                 7
|1BBBBBBBBBBBBBBBBB|
|                  |
```

where BBB...B is the debugging code of above.

Note that devadds in VTOCEs have no add_type: the add_type is strictly a page control concept. Any address in a VTOCE that is not a null address as above, i.e., has bit zero equal to zero, is a live secondary storage address-with the contents of the associated page out on it for a fact. That is the end result of the address management policy explained in Section VII. Such addresses have the format:

Format of a segment control device address, appearing only in a VTOCE file map:

```
 0                    1
 0                    7
|0DDDDDDDDDDDDDDDDDDDD|
|                     |
```

where DDD...D is a disk record address on the physical volume on which the VTOCE in which this address appears is found. See Section II for more information about addresses in VTOCEs.

PAGING DATA OBJECTS

Having described the critical concept of a devadd, we now describe the three paging data objects:

1. The PTW, representing a page of a segment, also being the hardware descriptor for that page.

2. The core map entry (CME), representing a page-frame of main memory and describing its association, if any, with any page of any segment.

3. The PDMAP entry, or PDME, describing a record of paging device, and its association, if any, with any page of any segment.

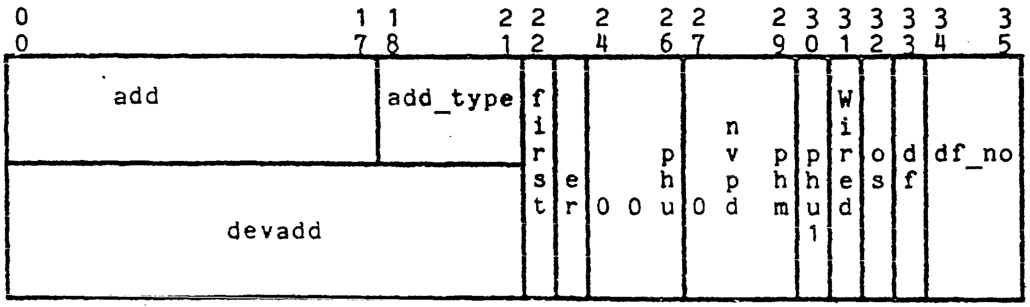
All of these data objects reside in the SST. All of them contain devadds as substructures. Many of these structures have fields that have different uses, and names, depending upon other bits and their meaning. The multiple names (e.g., cme.ptwp and cme.pdmep refer to the same storage) are used in the ALM include file. However, since this is impossible to describe in PL/I, the PL/I include files describe structures called "mpdme," "mptw," "mcme" to re-describe the structures for the alternate field names. In the descriptions below, we give the "alternate" PL/I names for the alternate fields, pointing it out when we do so with the warning "(Alternate for cme.xxx)". We give octal masks to help those interpreting dumps.

PTW, OR PAGE TABLE WORD

```
dcl 1 ptw based (ptp) aligned,

    (2 add bit (18),
    2 add_type bit (4),
    2 first bit (1),
    2 processed bit (1),
    2 pad1 bit (1),
    2 unusable1 bit (1),
    2 phu bit (1),
    2 unusable2 bit (1),
    2 nypd bit (1),
    2 phm bit (1),
    2 phu1 bit (1),
    2 wired bit (1),
    2 os bit (1),
    2 df bit (1),
    2 df_no bit (2)) unaligned;

dcl 1 mptw based (ptp) aligned,
    2 devadd bit (22) unaligned,
    2 pad bit (14) unaligned;
```



ptw.add
(777777,du)

When this PTW describes main memory, ptw.add is the upper 18 bits of the 24-bit main memory address of the main-memory page frame it designates. This can be the case whether or not ptw.df is on; only in the latter case is this PTW a valid hardware descriptor for the page; all other cases cause a process to take a page fault if it attempts to use this PTW as a hardware descriptor.

ptw.add_type
(740000,dl)

Defines which type of devadd is contained in this PTW; when it is add_type.core, 400000,dl, the field ptw.add is valid as above. Any type of page control devadd can appear here.

mptw.devadd
(777777740000)

(Alternate for ptw.add and ptw.add_type). Describes, if this page is in main memory, its main memory address, as a "main memory" type devadd. If this page is not in main memory, but is on the paging device, then this is a paging-device type devadd. If this page is neither in main memory nor the paging device, but has a disk record associated with it, this is a disk_type devadd as above, including a "nulled" bit on or off with the meaning explained. Otherwise, this is a true "null" page, and this is a null devadd as above. In all cases, this devadd designates the storage device or lack thereof from which the page will be read in or created if faulted on. A null address or a nulled address causes the creation of a page of zeros.

ptw.first
(200000,dl)

If the global switch sst.ptw_first is on, which it normally is not, pc$fill_page_table turns this bit on in all PTWs of segments being activated. This bit is turned off whenever this page is evicted from main memory. This bit being on tells the paging device allocator not to allocate a paging device record for this page when an attempt is made to evict it. Thus, if sst.ptw_first is on, paging device management is effectively changed so that pages get one chance to be referenced, in any given activation, and evicted, before being migrated to the paging device. This is desirable for random-access applications, to avoid suboptimal use of the paging device. An experimental feature, the flag sst.ptw_first may be set on only by highly privileged patching.

ptw.er,
ptw.processed
(100000,dl)

Used for two purposes. The interrupt side, when posting (telling the rest of page control about) the completion of a page read operation that was unsuccessful due to a device error, sets this bit, and notifies the faulting process. The restarted process takes the page fault over again, as the PTW has not been made to describe main memory (made valid as a hardware descriptor), notices this bit, turns it off so that the next process can retry this operation, and signals "page_fault_error" in that process. The post_purge service of page control uses this bit to mark all PTWs found in the PDS trace list (see Post Purge, in "Services of Page Control"). If any attempt is made to mark any PTW that has this bit on already, the implication is that the process has faulted on that page at least twice during its last eligibility and this is considered to be "thrashing"; the counter sst.thrashing is incremented. This bit is also used by online SST analysis tools (e.g., check_sst) to perform various marking operations on images of the SST.

ptw.phu
(001000,dl)

This bit is set to "1"b when the processor appending unit fetches this PTW, and places it into its associative memory. This page may be used repeatedly, but this bit will not be set again until that PTW leaves the processor's associative memory, either by replacement, or the execution of a CAMP instruction (clear PTW associative memory). The page replacement algorithm, in claim_mod_core, when noticing this bit and turning it off, does *not* clear the system's associative memories; it counts on the fact that some page eviction in the near future will. Clearing the associative memories of the system disturbs all processes and processors; the page replacement algorithm's approximations are not worth that much.

ptw.nypd
(000200,dl)

(Not yet on paging device.) This bit indicates that the page has been paged in from secondary storage, and has not yet migrated to the paging device. Thus, the main memory replacement algorithm is wary of evicting such pages, because it takes work (paging device writes) to do so. This bit is only meaningful when ptw.phm (see below) is zero for when the page has been modified in main memory, this alone is an indication to the main memory replacement algorithm that the page takes work to evict. Note that this bit shares a zone with ptw.phm; it does not matter that the appending unit modifies this zone when setting ptw.phm, as ptw.phm being on makes ptw.nypd meaningless.

ptw.phm
(000100,dl)

Page-has-been-modified bit. Set by the appending unit to "1"b when a reference is made to the page described by this PTW which stores into that page, and no PTW with the ptw.phm bit corresponding to this PTW appears in the associative memory. Therefore, when this bit is turned off by page control, the associative memories of the system processors must be cleared or future modifications may not be seen (see "write_page" in the "mechanisms" chapter). Such a store also turns on the ptw.phm bit in the PTW associative memory of the processor. Note that setting ptw.phm may affect ptw.nypd; this is a feature (see ptw.nypd above).

ptw.phu1
(000040,dl)

"Used in quantum bit." This bit is used only as input to the post-purge algorithm, which describes what to do with what pages, for performance reasons alone, at the end of a process' eligibility. This bit is turned on by the main memory replacement algorithm (claim_mod_core) every time ptw.phu is turned off, and is turned off by the post-purge algorithm under certain conditions. (See "Post-Purge" in Section IX.)

ptw.wired
(000020,dl)

Tells the main memory page replacement algorithm that this page may not be evicted under any circumstances, as some procedure is using it, or will use it, which may not take page faults. Such a page is said to be *wired*. Nevertheless, this page may be moved around main memory during reconfiguration operations, as long as it constantly remains accessible. (See "Eviction" in Section VIII), which is *not* true for an abs_wired page. All abs_wired pages are wired.

ptw.os
(000010,dl)

For "out of service." When *on*, an I/O operation is in progress on this page. Does *not* in general, mean that the page is inaccessible, or unusable in any way (pages are fully accessible during writes). When this bit is on, the "devadd" of the PTW must be a main-memory type devadd, describing a main memory address.

ptw.df
(000004,dl)

"directed fault" bit used by the hardware. When <u>on</u>, indicates that this PTW is a valid hardware descriptor, mapping references to some page of its segment into references to main memory. In this case, the "devadd" in the PTW must be a main-memory address, as ptw.add will be interpreted by the hardware as such. When <u>off</u>, a process attempting to use this PTW via the hardware will take a page fault. Note that processes will observe the fact that this bit has been turned off only if any copies of this PTW in their associative memories are cleared out; thus, all associative memories of the system are cleared when a page is evicted.

ptw.df_no
(000003,dl)

The contents of this field tell the hardware what type of directed fault to take when ptw.df indicates that it should take a fault. In Multics, this field is always set to "01"b, and thus, a directed fault 1 is interpreted as a Multics page fault. Note that zeros in a PTW, or an attempt to use zeros as a page table will <u>not</u> cause the page fault handler to be invoked, but rather the segment fault handler, for directed fault zero is interpreted as a segment fault (as uninitialized SDWs, which are in unused (zero) regions of descriptor segments, contain all zeros, specifically in sdw.df and sdw.df_no). This generally causes the segment fault handler to repeatedly issue the message "seg-fault: illegal segfault on CPU A" when it finds that the SDW contains no segment-fault condition at all.

<u>CORE MAP</u>

The Core Map is an array of Core Map Entries (CMEs), one for each page frame of configurable main memory. It is indexed by main memory address. The pointer sst.cmp points to the array, i.e., the CME for the frame at location 0. It is in the SST.

<u>CORE MAP ENTRY (CME)</u>

```
dcl 1 cme based (cmep) aligned,
    2 fp bit (18) unaligned,
    2 bp bit (18) unaligned,

    2 devadd bit (22) unaligned,
    2 padding bit (2) unaligned,
    2 io bit (1) unaligned,
    2 rws bit (1) unaligned,
    2 er bit (1) unaligned,
    2 removing bit (1) unaligned,
    2 abs_w bit (1) unaligned,
    2 abs_usable bit (1) unaligned,
    2 notify_requested bit (1) unaligned,
    2 spare bit (2) unaligned,
    2 contr bit (3) unaligned,

    2 ptwp bit (18) unaligned,
    2 astep bit (18) unaligned,
    2 dblw_devadd bit (22) unaligned,
    2 padding1 bit (14) unaligned;

dcl 1 mcme based (cmep) aligned,
    2 pad bit (36) unaligned,
    2 record_no bit (18) unaligned,
    2 add_type bit (4) unaligned;
```

```
            0            1 1              3
            0____________7_8______________5
            |             |               |
Word 0      |     fp      |      bp       |
            |_____________|_______________|
            0               2 2     3 3   3
            0_______________1_2_____2_3___5
            |               |       |     |
Word 1      |    devadd     | flags |ctrlr|
            |_______________|_______|_____|
            0            1 1              3
            0____________7_8______________5
            |             |               |
            |    ptwp     |               |
Word 2      |_____________|     astep     |
            |    pdmep    |               |
            |             |               |
            |_____________|_______________|
            0                             3
            0_____________________________5
            |                             |
Word 3      |          reserved           |
            |_____________________________|
```

cme.fp
(777777000000,word 0)

Forward pointer along with cme.bp, defines the position of the CME in the core map used list, used by the main-memory page replacement algorithm to maintain pseudo-LRU order. The rel-pointer cme.fp is the relative offset into the SST of that CME which describes the page frame containing the page supposedly slightly more recently seen as used. Its field cme.bp describes this CME. (See "Main Memory Replacement Algorithm" in Section V.) When a page-frame is undergoing either an I/O operation, reading or writing a page, or an RWS (cme.rws on), both cme.fp and cme.bp are zero, and no other CME, or either of the used-list pointers, sst.usedp and sst.wusedp, designate this CME. The fields cme.fp and cme.bp are both "777777"b3 in CMEs that designate pages that are not configured, or are deconfigured. CMEs not part of the paging pool, but still corresponding to real main memory, are all zeros.

cme.bp
(000000777777,word 0)

Back pointer. See cme.fp above.

cme.devadd
(777777740000,word 1)

A devadd as described in the beginning of this section. Valid only when cme.ptwp (or mcme.pdmep) is nonzero. May only validly be a paging device address, or nulled or live disk address. If cme.rws is off, then this is that address to which the page whose PTW is described by cme.ptwp will be written when evicted; a paging device devadd if this page has one, otherwise a disk address. If cme.rws is on, i.e., an RWS is in progress in this main memory frame, the contents of cme.devadd depend upon cme.io, which tells whether the read or write half of the RWS is under way, and the paging device or disk address resides here respectively.

cme.flags
(000000037770)

Various state flags, detailed below.

cme.io
(004000,dl)

Valid only if cme.ptwp (or mcme.pdmep) is nonzero. Tells the direction of I/O if any is going on in this frame, off being read, on being write. Valid as above, and at that, only if:

If cme.rws is on, tells whether a Read or Write cycle of an RWS is in progress here.

If cme.rws is off, then the PTW designated by cme.ptwp must have ptw.os on if cme.io is meaningful, in which case that page is being read or written from this main memory frame, and cme.io tells which. Basically tells the interrupt side what to do.

cme.rws
(002000,dl)

Valid only when mcme.pdmep is nonzero (if cme.ptwp describes a PTW, page control is in a severe error situation. This bit being on, when mcme.pdmep is nonzero, means that an RWS is going on in this main memory frame. The flag cme.io tells which half of the RWS; mcme.pdmep contains the relative offset into the SST of the PDMAP entry for the paging device record undergoing RWS. It must have pdme.rws on, and be out of the PDMAP used lit. This CME must be out of the used list.

cme.er
(001000,dl)

is NOT USED.

cme.removing
(000400,dl)

is turned on by pc_abs on the call side when the main memory page frame described by this CME is being deconfigured. It makes find_core skip over this page, ensuring that any eviction from this page frame is permanent until the page frame is threaded out of the used list, making it totally inaccessible. (See "Main Memory Deconfiguration Service" under "Services" in Section IX.)

cme.abs_w
(000200,dl)

Defines a page frame containing an "abs-wired" page, or a page frame in the process of receiving such a page. Such a page will also be marked as "wired" in its PTW. Keeps find_core from trying to evict the contents of this page, or handing it to any caller of find_core during interim states (such as possible FSDCT pagings) during the wiring of this page when the page frame might otherwise appear to be free. Also informs the main memory configuration service that the controller containing this page frame cannot be deleted. Also informs the allocator of abs-wired main memory that this page frame is already abs-wired, and its contents cannot be moved to make room for abs-wired pages. (See "Abs Wiring Service" in Section IX.)

cme.abs_usable
(000100,dl)

Says that this page frame may, if not already used so, be used for abs-wiring, if this page frame is usable (appears in the used list or is actually in use) at all. All page frames with cme.abs_w on must have cme.abs_usable on. This quality of being abs-usable is a static function of a page frame throughout a bootload. See the Multics Reconfiguration PLM, Order No. AN71.

cme.notify_requested
(000040,dl)

Valid only if cme.rws is off, and cme.ptwp describes a CME with ptw.os on (in which case this CME is threaded out of the used list, as a page I/O is in progress). Tells the interrupt side that some process is waiting, via the traffic controller wait/notify mechanism for I/O completion on this page. This bit is turned on when any process goes to wait for paging I/O, either on the fault side (see "Page Fault Handling" in "Services,") the call side, via the call-side wait coordinator, device_control$pwait (see "Wait Protocols" in "Mechanisms"), or the special wait mechanism of the process-loading mechanism (see "Process Loading" in "Services"). It tells the interrupt side to invoke the traffic controller to perform a "notify" on the event associated with this page (see "Wait Protocols" in Section VIII) when the I/O on this page is complete. If not on, no traffic control notify is performed when this I/O completes.

cme.pd_upflag
(000020,dl)

Causes the interrupt side to rethread this CME to most recently used position on the completion of a page write from this frame, as opposed to the least recently used position as it normally does.

cme.contr

(000007,dl)

Not currently used. (Controller) is the port tag of the system controller that controls the main memory described by this CME. (See the Multics Reconfiguration PLM, Order No. AN71.)

cme.ptwp
(777777000000,word 2)

PTW pointer. Only valid when cme.rws is off. When nonzero, states that some page of some segment is associated with this page frame. The field cme.ptwp is the relative offset into the SST of the PTW for that page. The page may or may not be undergoing I/O as ptw.os of that PTW is on or off. The page is not, however, undergoing RWS. It is guaranteed that the "devadd" file of the PTW has a main-memory type devadd describing the main memory page frame of this CME.

mcme.pdmep
(777777000000,word 2)

(Alternate for cme.ptwp). Only valid when cme.rws is on, which is when there is an RWS going on in this main memory frame. In this case, mcme.pdmep is the relative offset into the SST of the PDMAP entry of the PD record undergoing this RWS. In this case, the field mpdme.cmep of that PDME would be the relative offset into the SST of this CME.

cme.astep
(000000777777,word 2)

Only valid under the conditions under which cme.ptwp is valid and nonzero. The field cme.astep will then contain the relative address into the SST of the AST entry for the segment to which the page in this main memory frame belongs.

Word 3 of the core map entry is reserved for future expansion. It is no longer used as "cme.dblw_devadd."

PAGING DEVICE MAP

The Paging device map is an array of Paging device map entries (PDMEs), one for each configurable record in the Paging device. It contains PDMEs for all PD records to be used by the current bootload, as specified by the PAGE CONFIG card. The pointer sst.pdmap located the PDME for record 0 of the paging device. It is in the SST.

PAGING DEVICE MAP ENTRY (PDME)

```
dcl 1 pdme based (pdmep) aligned,
    2 fp bit (18) unaligned,
    2 bp bit (18) unaligned,

    2 devadd bit (22) unaligned,
    2 pad2 bit (2) unaligned,
    2 modified bit (1) unaligned,
    2 incore bit (1) unaligned,
    2 rws bit (1) unaligned,
    2 used bit (1) unaligned,
    2 abort bit (1) unaligned,
    2 pad3 bit (1) unaligned,
    2 flushing bit (1) unaligned,
    2 notify_requested bit (1) unaligned,
    2 update_only bit (1) unaligned,
    2 removing bit (1) unaligned,
    2 double_writing bit (1) unaligned,
    2 pad bit (1) unaligned,

    2 ptwp bit (18) unaligned,
    2 pageno fixed bin (8) unal,
    2 pvtx  fixed bin (8) unal,

    2 uid bit (36) aligned;

dcl 1 mpdme based (pdmep) aligned,
    2 save_old_pvtx fixed bin (17) unaligned,
    2 cmep bit (18) unaligned,
    2 record_no bit (18) unaligned,
    2 add_type bit (4) unaligned;
```

This page intentionally left blank.

```
          0              1 1               3
          0              7 8               5
         ______________________________________
         |               |                    |
         |      fp       |         bp         |
Word 0  _|_______________|____________________|
         | save_old_pvtx |        cmep        |
         |               |                    |
         ______________________________________
          0                 2 2            3
          0                 1 2            5
         ______________________________________
         |                  |                 |
Word 1   |     devadd       |      flags      |
         |                  |                 |
         ______________________________________
          0              1 1       2 2     3
          0              7 8       6 7     5
         ______________________________________
         |               |         |          |
Word 2   |     ptwp      | pageno  |  pvtx    |
         |               |         |          |
         ______________________________________
          0                                3
          0                                5
         ______________________________________
         |                                    |
Word 3   |               uid                  |
         |                                    |
         ______________________________________
```

pdme.fp
(777777000000,word 0)

Forward pointer in the PD used list. Has the relative address into the SST of the PDME used supposedly slightly more recently than this one. PDMEs describing records that are undergoing RWS are threaded out: pdme.fp is zero, and pdme.bp is reused as mpdme.cmep. PDMEs that have been deconfigured have pdme.fp and pdme.bp both equal to "777777"b3. Paging device map entries in PDMAPs representing "unflushed" paging devices, on the next bootload after one in which ESD failed, have all entries either threaded out or deconfigured. This field shares storage with mpdme.save_old_pvtx.

mpdme.save_old_pvtx
(377777,du,word 0)

(Alternate for pdme.fp.) During a post-crash PD flush, the value of pdme.pvtx is saved here. This is so that should the system crash during the post-crash PD flush, the next bootload can put that PVT index back in pdme.pvtx to retry the flush. The field pdme.pvtx is set, during the post-crash flush, to the PVT index of the drive where the volume to which the pages are being repatriated in this bootload. The old value is necessary to identify the pack, where it was recorded in the label at the time the volume was accepted (see "Post-Crash PD Flush" under "Services," and Section IX.)

pdme.bp
(000000777777,word 0)

Backward pointer in the PD Used list. Has the relative offset of the PDME, in the SST, whose pdme.fp describes this pdme. Also shares storage with mpdme.cmep. Valid only when pdme.rws is off.

mpdme.cmep
(000000777777,word 0)

(Alternate for pdme.bp.) Valid only when pdme.rws is on, in which case pdme.fp should be zero and no other PDME or the PD used list used pointer sst.pdusedp should describe this PDME. In this case, an RWS is being undergone by the PD record described by this PDME, and mpdme.cmep contains the relative address in the SST of the CME that describes the page frame in which this RWS is taking place. The field mcme.pdmep should point back to this PDME. Used by the abort code in the interrupt side to locate the CME when the PDME has been found from the PTW. See Figure 6-5.

pdme.devadd
(777777740000,word 1)

Is the disk address, as a standard page control devadd, which is associated with the page contained on the PD record described by this PDME (valid only when pdme.used is on). Must be a disk-type devadd, can be nulled or live. Pages created in main memory, written to the paging device, but never yet written to the disk record which they were assigned will have a nulled devadd here (see "Address Management," Section VII).

pdme.flags
(037777,dl,word 1)

Are the pdme control flags, detailed below.

pdme.mod
(004000,dl)

Modified with respect to disk. Indicates that the page in the PD record described by this PDME is different from the copy of the page, if any, on disk, and an RWS will be necessary to free this PDME.

pdme.incore
(002000,dl)

Is OBSOLETE. PTWs are inspected directly by the paging device replacement algorithm.

pdme.rws
(001000,dl)

If on, the record of paging device described by this PDME is undergoing RWS. The CME designated by mpme.cmep contains additional information. See the description of that field above.

pdme.used
(000400,dl)

Indicates, when on, that this pdme is not free, i.e., that the PD record it describes contains some page of some segment. All fields other than the thread word of a PDME are zeros when it is freed, unlike CMEs. The bit pdme.used being off in a nonzero PDME should not validly occur.

pdme.abort
(000200,dl)

Turned on by the fault side when this function discovers that an RWS is in progress on the PD record that contains the page it is trying to read in. This tells the interrupt side, upon completion of the RWS, to connect the PTW to the main memory frame in which the RWS was performed, thus effectively paging the page in "by virtue of RWS," and not to free either the page frame or the PD record. It also causes the interrupt side to notify the RWS completion event (see "Wait Protocols" in Section VIII) to restart the faulting process.

pdme.flushing
(000040,dl)

Is used by the post-crash software when repatriating a page at volume-salvage time, after an unsuccessful shutdown. Turned on when the RWS for this page is initiated. Function is to tell the interrupt side that this is not an ordinary RWS, and the PDME should not be freed upon completion, but left intact so that the post-crash repatriator (pc$flush_seg_old_pd) can determine the relative success of the RWS by inspecting the PDME. (See "Post-Crash PD Flush" in Section IX.)

pdme.notify_requested
(000020,dl)

Parallel in function to cme.notify_requested. Turned on by the call-side wait coordinator, device_control$pwait, when the call side wants to wait for the completion of an RWS. Tells the interrupt side to perform a traffic control "notify" on the RWS event for this PDME. Note that this is always done for an RWS abort completion, which is when the same thing happens on the fault side.

pdme.update_only
(000010,dl)

Is OBSOLETE.

pdme.removing
(000004,dl)

Is used during deconfiguration of the entire, or partial paging device, by the operator "delpage" command. Useful only during an RWS, it tells the interrupt side, on completion of the RWS, not to free the PDME, but to deconfigure (delete) it. Also used internally by the interrupt-side automatic deconfiguration code which responds to paging device errors (see "Error Handling" in "Mechanisms").

pdme.double_writing
(000002,dl)

Used when the paging device is being used in any of the double-write (write-through) modes specifiable by the PARM DBLW parameter in the CONFIG deck. This bit is turned on by the interrupt side upon the completion of a paging device write if it is decided that a double-write to disk will be performed. This decision is made based upon the number following the word DBLW on the PARM card, and the properties of the page just written. It is on while the double-write (to disk) is going on. It tells the interrupt side, upon completion of the write, that the page has been successfully written to disk, and therefore, that the disk address in the PDME (pdme.devadd) should be resurrected. (See "Address Management," in Section VII.)

pdme.ptwp
(777777000000,word 2)

Is a pointer, relative to the SST, of the PTW for the page that resides on the PD record described by this PDME. In the case where the contents of the paging device are left over from a previous bootload, which did not shut down successfully, pdme.ptwp is zero, until the paging device is reinitialized when it is successfully flushed. The fact that this field is always nonzero during normal operation is a reflection of the policy that only pages of active segments are allowed on the paging device.

pdme.pageno
(377000,dl)

Along with pdme.pvtx and pdme.uid, this field is there principally for the post-crash PD flush done by the next bootload after a crash in which ESD did not succeed. The field pdme.pageno is the page number, relative to zero, within its segment, of the page on this record of paging device.

pdme.pvtx
(000377,dl)

The index in the physical volume table of the drive which contains that pack, on which the page in the PD record described by this PDME resides. This field is used by the interrupt side, at the mid-point of an RWS, to identify the drive to which the RWS buffer must be written for the write cycle of the RWS. (See "Post-Crash PD Flush," Section IX.)

pdme.uid
(whole word 3)

Is the unique segment ID of the segment containing the page that resides in the PD record described by this PDME. This is placed here by the PD allocator, allocate_pd in page_fault, solely so that this PDME can be "found" during physical volume salvaging of the pack containing that page, so that this page might be repatriated at that time.

PDMAP HEADER

The PDMAP header occupies that region of the paging device map which would otherwise be the PDME for the first record used. Since this record is always guaranteed to contain a copy of the first page of the PDMAP, the space is used for the PDMAP header. (See "Post-Crash PD Flush" in Section IX for motivation for the PDMAP header.) Other than pdmap_header.time_of_bootload, the PDMAP header contains copies of similarly-named information in the SST.

```
dcl 1 pdmap_header based (pdmhp) aligned,
    2 pd_first fixed bin (17) unal,
    2 pd_using fixed bin (17) unal,
    2 nrecs_pdmap fixed bin (17) unal,
    2 pdme_no fixed bin (17) unal,
    2 time_of_bootload fixed bin (71);
```

pdmap_header.pd_first

Copy of sst.pd_first. The paging device record number or the first record being used by this bootload; this first record is the one containing the first record of the PDMAP.

pdmap_header.pd_using

Copy of sst.pd_using. The number of records of the paging device usable as a paging device--includes all those in use or free. Does not include those deconfigured or used to store the PDMAP.

pdmap_header.nrecs_pdmap

Copy of sst.nrecs_pdmap. The number of pages (1024-word lengths) in the length of the PDMAP itself; the number of bulk store records devoted to storing the map itself.

pdmap_header.pdme_no

Copy of sst.pdme_no. The number of elements in the PDMAP array, including those corresponding to records in which the copy of the PDMAP is stored on the bulk store.

pdmap_header.time_of_bootload

The value of fsdct.time_of_bootload (always set to the clock during collection 1 initialization) from that Multics bootload during which this instance of the paging device map was initialized. This quantity will not change during successive bootloads after a crash in which ESD fails, until all pages on the paging device have been repatriated, at which time the PD map will be reinitialized. This quantity is written to the labels of all physical volumes (label.pd_time) accepted during a bootload in which this PDMAP was actively in use; this allows the post_crash PD flush to identify those volumes to which pages need to be repatriated.

PVTE VARIABLES FOR PAGE CONTROL

The PVT, or physical volume table, is basically a data base of volume management. However, it contains in its PVTEs (PVT entries) all of the per-drive and per-mounted-pack data used by the system, specifically the information used by the disk DIM to describe a drive, and the information used by the disk record allocator/deallocator (free_store) of page control. All of the following parameters are used by the disk record allocator/deallocator; the other parameters in the PVTE are described in Section XIII. These parameters describe the status of the bit-map of free records for that volume. Historically, these parameters had lived in the FSDCT, in a region directly preceding the bit-map, and were known as fsmap parameters. (See "Disk Record Allocation/Deallocation" in "Mechanisms.")

pvte.fsmap_rel

a relative pointer, relative to the base of the FSDCT, to the bit map for this drive.

pvte.curwd

a relative pointer, relative to the base of the bit map for this drive, of the next word to be inspected for free records.

pvt.wdinc

a number by which pvte.curwd is to be incremented to "roll it around" to the beginning when it passes the end of the bit-map.

pvte.temp

is a temporary variable used as such by free_store. This highly unlikely place for a work variable is historical in origin.

pvte.baseadd

is the record address represented by the first bit of the bit-map for this drive. Each word represents 32 addresses, starting at that record address. The first bit of each word is not used, nor are the last three bits. This is to facilitate assembler-language manipulation of this table.

pvte.tablen

is the number of valid words, for the pack currently mounted on this drive, of the bit-map.

pvte.tablen_allocation

is the number of words in the FSDCT region allocated for this drive. This is a function of the drive, not the pack on it.

pvtw.nleft

is the number of bits on at any time in the bit-map for this drive, i.e., the number of records left unallocated. When zero, an "out of physical volume" (OOPV) situation has occurred.

pvte.relct

is a counter of the number of deposits (freeings) performed since last reset. When this number reaches 100, it is reset, and pvte.curwd reset to the beginning of the free store map.

pvte.totrec

is the number of records described by the bit-map for this pack.

SYNOPSIS OF RELEVANT SST VARIABLES

The SST header, the first 512 words of the SST, contains a large number of global variables of interest to the storage system in all its subsystems. However, the large number of them which directly control every action of page control make it mandatory to list these variables, and give their interpretations.

sst.space

first eight words of SST. Set to "777777777777"b3 by init_sst. Used to watch for page control bugs which might accidentally use zero rel-pointers, and thus store data intended for somewhere else into the first few words of the SST.

sst.post_purge_time

a cumulative total of CPU time spent in the post-purge function. Reported by post_purge_meters.

sst.post_in_core

a count of pages found in main memory by the post-purge function at post-purge time. Indicative of working-set behavior.

sst.thrashing

a count of pages found twice in a per-process page-trace list by the post-purge function. Indicates that a process could not even keep its working set in main memory during its eligibility.

sst.npfs_misses

is OBSOLETE.

sst.salv

is OBSOLETE.

sst.ptl

is the actual global page table lock.

sst.nused

is the number of page-frames of main memory in use by paging, be they wired, out of service, free, or whatever. Pages deconfigured, not corresponding to real memory, or containing parts of perm-wired segments are not counted. Critical for the traffic controller's memory-sharing computations.

sst.ptwbase

is the absolute address of the base of the SST segment. Used to convert SST-relative page-table pointers into absolute addresses suitable for use in SDWs, and vice-versa.

sst.bulk_pvtx

is the PVT index of the bulk store. The bulk store has a PVT entry, and is therefore, in some contexts, considered a rather peculiar type of disk. Specifically, it is that "disk" on which the "pdmap_seg," the segment that is used to access and update the PDMAP image on the bulk store, resides.

sst.astsize

is 12 decimal, the size of an AST entry.

sst.cmesize
is 4, the size of a CME.

sst.cmp
is an ITS pointer to the base of the core map array, which is always the CME for address zero.

sst.usedp
is a relative pointer to the CME which is the best candidate for replacement. This field is the "clock-hand" of the main memory page replacement algorithm.

sst.wtct
is a count of all outstanding writes initiated by page control. When this number reaches a certain threshold (a "ceiling" is then said to have occurred) the DIMs are interrogated for completions until this number goes down. (This is called "running the devices," see "Mechanisms.")

sst.startp
is OBSOLETE.

sst.removep
is OBSOLETE.

sst.double_write
is the parameter that appears on the PARM DBLW CONFIG card field, if there is one, otherwise zero. It tells the paging device interrupt side when, if at all, to perform double-writes, based upon its value:

0 Never double write, the default.
1 Double write every time a PD write is done, but not process directory pages.
2 Double write only directory pages.
3 Double write anything which has never been double-written, i.e., needs resurrection.

sst.temp_w_event
is "200000000000"b, used by wire_proc to lock the "temp-wiring" tables. (See Section X.)

sst.root_pvtx
is the PVT index of the RPV (Root Physical Volume), on which all of the supervisor resides, and the whole system runs during initialization.

sst.ptw_first
if patched on, modifies paging device behavior to give all pages a chance to be used and evicted once before migrating them to the paging device. (See the description of ptw.first, earlier.)

sst.nolock
is OBSOLETE.

sst.x_fsdctp
is OBSOLETE.

sst.pdir_page_faults
is a meter of page faults on per-process segments. Reported by file_system_meters.

sst.level_1_page_faults
is a member of page faults on directories and segments off of the root. Reported by file_system_meters.

sst.dir_page_faults
is a meter of page faults on directories. Reported by file system meters.

sst.ring_0_page_faults
is a meter of page faults taken in ring zero. Reported by file_system_meters.

sst.rqover
is the value of error_table_$rqover, the error code for record quota overflow. Put here so that the page-fault handler can use it, as it cannot reference error_table_, the latter not being wired.

sst.pc_io_waits
is OBSOLETE.

sst.steps
is the number of times the main memory page replacement algorithm (see the earlier description) passed a CME. Reported by file_system_meters.

sst.needc
is the number of times the main memory page replacement algorithm was invoked, i.e., a page frame was needed. Reported by file_system_meters.

sst.ceiling
is the number of times the page replacement algorithm had to "run the devices" because of an excess of writes queued. (See "sst.wtct" above.) Reported by file_system_meters.

sst.ctwait
is OBSOLETE.

sst.wired
is a count of the number of pages temp-wired or abs-wired.

sst.laps
is OBSOLETE. File_system_meters computes "laps" as "steps" divided by "nused."

sst.skipw
is the number of times the main memory PRA skipped page frames containing abs-wired or temp-wired pages. Reported by file_system_meters.

sst.skipu
is the number of times that the main memory page replacement algorithm passed over a page because it was recently used, and turned off its "used" bit. Reported by file_system_meters.

sst.skipm
is the number of times that the main memory page replacement algorithm skipped a page because it was modified, and needed writing out. Reported by file_system_meters.

sst.skipos
is OBSOLETE.

sst.skipspd
is OBSOLETE.

sst.reads
is an array by device type, metering read requests dispatched by device_control$dev_read for each type of device.

sst.writes
is an array, by device type, metering write requests dispatched by device_control$dev_write, for each type of device.

sst.short_pf_count
is a count of the number of times that a page fault had already been satisfied (usually by some other process) by the time it successfully locked the page table lock.

sst.loop_locks
is a count of attempts to lock the page table lock.

sst.loop_lock_time
is a cumulative total of CPU time spent looping on the page table lock. It is reported by total_time_meters.

sst.pre_page_size
is OBSOLETE.

sst.post_list_size
is a count of all page trace entries processed by the post-purge function (see Section IX). When divided by sst.post_purge_calls, it is the average size of the post-purge list.

sst.post_purgings
is a count of all page writes started by the post-purge function, which is an option currently not selected (see Section IX).

sst.post_purge_calls
is a count of invocations of the post-purge function.

sst.pre_page_calls
sst.pre_page_list_size
sst.pre_page_misses
sst.pre_pagings
all are OBSOLETE.

sst.wire_proc_data
is used solely by the procedure wire_proc (see Section X, "Peripheral Services of Page Control") to keep track of temp-wiring requests.

sst.abs_wired_count
is a count of all page frames containing abs-wired pages.

sst.wired_copies
is OBSOLETE.

sst.recopies
is a count of the number of times that evict_page had to recopy a page because it was modified while being copied. (See "Demand Eviction" in Section VIII.)

sst.first_core_block
is zero.

sst.last_core_block
is the index in the core map of the highest-addressed page frame in the configuration. Used by reconfiguration (see the Multics Reconfiguration PLM, Order No. AN71).

sst.tree_count
is an array of sixty-four cells, corresponding to the sixty-four possible page-states which the post-purge function can see. It counts how many times each was encountered. (See Section IX, "Post Purging.")

sst.pp_meters
is OBSOLETE.

sst.wusedp

is the "write" usedp, used by claim_mod_core to do writes and PD migrations until it is equal to sst.usedp. (See "Main Memory Replacement Algorithm" in Section V.)

sst.write_hunts

is the number of times that claim_mod_core was invoked to do work postponed by find_core.

sst.claim_skip_cme

is the number of times that claim_mod_core attempted to process a CME which was unprocessable, i.e., was abs-wired.

sst.claim_skip_free

is the number of times that claim_mod_core passed over a CME which was free. As the region of the list being processed by claim_mod_core is directly behind usedp, this is not a good state of affairs; that CMEs should be at the other end of the list.

sst.claim_notmod

is a meter on the number of times that claim_mod_core passed a page that was not modified or "nypd," and thus not even interesting.

sst.claim_passed_used

is a count of times that claim_mod_core passed pages whose "used" bits were on, turning them off on behalf of find_core.

sst.claim_skip_ptw

is a meter on the number of times that claim_mod_core passed a page and skipped it because of the state of its PTW; usually, this means that the page was wired.

sst.claim_writes

is a count of calls made by claim_mod_core to write out pages (if full of zeros, the pages will not actually be written).

sst.claim_steps

is a count of core map entries processed by claim_mod_core.

sst.rws_reads_os

is a count of outstanding RWS "read" cycles (paging device read) in progress. The RWS initiator of the paging device replacement algorithm initiates all of the RWSs it is going to at once, and waits for sst.rws_reads_os to become zero via "running" the bulk store DIM. While allowing the full queueing facility of the bulk store to be used, this ensures that the page table is not unlocked during RWS read cycles, as page control is not prepared to handle aborts during the read side.

sst.pd_updates

is a count of done-time PD writes started, part of the feature described under sst.pd_writeahead.

sst.pre_seeks_failed

is a count of the number of times that find_core could not find an acceptable (not used, not modified, not "nypd," not wired) CME in fifteen steps, and called claim_mod_core as a result to cause more processing, to cause completions to be noticed and zero pages to be discovered.

sst.pd_desperation_steps

is a count of steps made by the PD desperator, which is invoked when the PD allocator finds that the PDME at the head of the PD used list is not claimable. The counter of failures of the PD desperator is sst.pd_no_free.

sst.pd_desperations

is a meter of the number of times the PD desperator was invoked (reported by page_multilevel_meters).

sst.skips_nypd

is a meter of times that the main memory replacement algorithm skipped a page frame because of its "not-yet-on-paging-device" status.

sst.pd_writeahead

is a flag used to enable an unsuccessful experiment which caused the paging device to be updated at disk-read completion time. This flag causes the PD allocator to inform the interrupt side to start a PD write, as opposed to turning on ptw.nypd, which is its normal action in this circumstance.

sst.pd_desperations_not_mod

is a count of the number of times that the PD desperator was invoked on behalf of a pure page, i.e., one which is an identical copy of a page on disk. Reported as a percentage of desperations by page_multilevel_meters.

sst.resurrections

is a count of the number of times that a disk devadd was resurrected, i.e., made non-nulled and thus reportable to segment control, by virtue of a disk write from main memory. (See Section VII, "Address Management Policy.")

sst.fsdct_oocore

is a count of "recursive" simulated pagings of the FSDCT done by the page fault handler to satisfy a need of allocating a disk record for the page being faulted on. (See "FSDCT Paging," Section VIII.)

sst.oopv

(Out of Physical Volume) is the number of times that page control, when invoked to allocate a disk record by the page fault handler, could not, because there were no more available. The only permissible circumstance is for a hierarchy segment, in which case, the SDW for the segment is faulted, provoking a segment move (see "Segment Moving" in Section IV).

sst.fsdct_ptp

is an ITS pointer to the page table of the FSDCT. This is needed by the "recursive" page fault simulator used to access the FSDCT during a page fault. (See "FSDCT Paging," Section VIII.)

sst.pd_resurrections

is a count of the number of times that a disk devadd was resurrected (see sst.resurrections above) by virtue of the successful completion of an RWS.

sst.dblw_resurrection

is a count of the number of times that a disk devadd was resurrected by virtue of the completion of a write-through from the paging device. (See sst.double_write.)

sst.pdflush_replaces

is a count of the number of times that the post-crash PD flush actually changed a disk address in a file map by virtue of this repatriation.

sst.pdmap

is a pointer to the virtual origin of the paging device map array, null if there is no paging device. Note that this not the first record being used, but rather, record zeros PDME, even if the place where that would be below the base of the SST.

sst.pdhtp

is OBSOLETE.

sst.pd_id
: is the PVT index of the device (the bulk store) which is the paging device. It is zero if there is no paging device (this is not the case when there is an unflushed paging device). (See "Post-Crash PD Flush," Section IX.)

sst.pdsize
: is 4, the size of a PDME in words.

sst.pdme_no
: is the number of elements in the PDMAP, i.e., the number of records in the region being used, including those being used to hold the copy of the PDMAP itself.

sst.pdusedp
: is the "clock hand" of the PD replacement algorithm. Contains the SST-relative address of the PDME at the "best candidate for replacement" (head) end of the PD used list. If there are any free PDMEs, they are right there.

sst.pd_first
: is the PD record number of the first record in the region of the paging device being used, the first number on the PAGE CONFIG card. This record number will be the one used to hold the first record of the PDMAP.

sst.pd_map_addr
: is the absolute main memory address of the base of the PDMAP in the SST segment. This is used by the function in check_pd_free_and_update in pd_util which invokes the bulk store DIM every second to write out the PDMAP to the first records of the bulk store.

sst.nrecs_pdmap
: is the number of records on bulk store occupied to hold the paging device map image.

sst.pd_free
: is the number of PD records either free or undergoing RWS; used by the PD replacement algorithm to free more or start more RWSs when this number sinks below 10.

sst.pd_using
: is the number or PD records either usable or being used to contain pages, i.e., not those which are deconfigured or contain the PDMAP image. When zero, this cell is an indication to all of page control that the paging device is not enabled (may be all deconfigured, or unflushed), and no PD migrations can or will be performed.

sst.pd_wtct
: is the total number of RWSs outstanding. The paging device replacement algorithm will not let this number get above thirty; if this threshold is reached, it loops "running" the DIMs until pd_wtct goes down. (See "DIM Interface," Section VIII.)

sst.pd_writes
: a counter of the number of RWSs ever initiated. Reported by page_multilevel_meters.

sst.pd_ceiling
: the number of times sst.pd_wtct hit thirty, and the paging device replacement algorithm had to loop.

sst.pd_skips_incore
: total number of times that the paging device replacement algorithm skipped over a PDME, rethreading it to "recently used" because it contained a page that was also in main memory at the time. (See "Paging Device Management Algorithm" earlier.)

sst.pd_skips_rws
is OBSOLETE.

sst.mod_during_write
is a counter of the number of times that a page being written out was found to have been used while being written. Indicates that the replacement algorithm made a poor choice.

sst.pd_write_aborts
is a count of RWS aborts performed, i.e., times when a page fault occurred on a page that was undergoing RWS. (See "Paging Device Management Algorithm" earlier.)

sst.pd_rws_active
is OBSOLETE.

sst.pd_no_free
is a count of times that the PD Desperator failed. (See "sst.pd_desperations" above.)

sst.pd_read_truncates
is OBSOLETE.

sst.pd_write_truncates
is OBSOLETE.

sst.pd_htsize
is OBSOLETE.

sst.pd_hash_mask
is OBSOLETE.

sst.pdmap_astep
is an ITS pointer to the AST entry of the hardcore segment "pdmap_seg," which is used by the call side to perform explicit readings and writings of the PDMAP image areas on the bulk store.

sst.zero_pages
is a count of the times that write_page, the page-writing primitive, found a page all full of zeros, and thus nulled its disk address instead of writing it out.

sst.pd_zero_pages
is a count of times that write_page performed the above service (see sst.zero_pages), and a copy of the page existed on the paging device, which caused the PD record to be freed.

sst.trace_sw.pc_trace
enabled via the hardcore trace facility, and switch 34 on the processor, causes page control to print out a large amount of debugging information as it proceeds, mostly obsolete.

sst.rws_time_temp
is a temporary used by the RWS initiator and the interrupt side to meter CPU time overhead of page multilevel.

sst.rws_time_start
a cumulation of CPU time spent in the RWS initiator. Printed out by page_multilevel_meters.

sst.rws_time_done
a cumulation of CPU time spent in the interrupt side processing RWSs. Printed out by page_multilevel_meters.

sst.pd_time_counts
is OBSOLETE.

sst.pd_time_values
is OBSOLETE.

sst.pd_no_free_gtpd
: is a meter of the number of times that the PD allocator did not migrate a page to the paging device because it belonged to a segment with the "Global Transparent Paging Device" attribute defined in Section II. Note that the PD allocator is invoked both at read-done time and at page-write time.

sst.pd_page_faults
: is a count of page faults from the paging device. Reported as a percentage by page_multilevel_meters.

sst.pd_no_free_first
: is a count of times that the PD allocator refused to migrate a page to the paging device because ptw.first was on, i.e., the feature described under "sst.ptw_first" thought that the page should not be so migrated.

sst.update_index
: is used by the periodic PDMAP writer in pd_util to keep track of which page of the PDMAP it is writing out.

sst.last_update
: is the clock time at which the PDMAP was last written out. If the current time, at the beginning of any page fault, is more than a second past this time, it is written out again.

sst.count_pdmes
: when set to 1 by patching, enables an experimental meter which meters, into sst.buckets, the depth of PDMEs in the PDME used list, at the time that they are rethreaded to the head. For the use and significance of this type of meter, see the paper by Greenberg cited in Section V. This meter is referred to there as the "Experiment of webber and Snyder." Enabling this meter engenders substantial overhead in the page-fault path, and should not be done frivolously.

sst.bucket_overflow
: is a count of times that the meter described under "sst.count_pdmes," above metered a rethreading so deep that it could not be metered in sst.buckets.

sst.buckets
: (See sst.count_pdmes.)

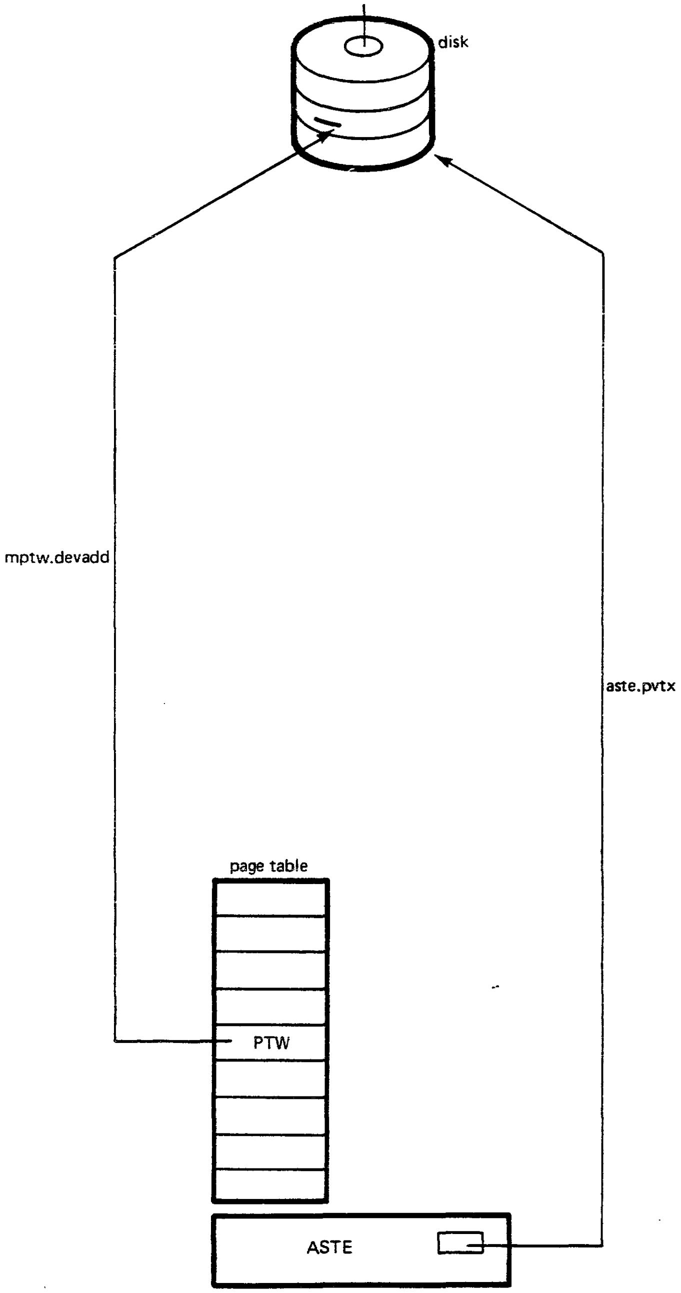


Figure 6-1. Page Control Data Bases
Page not in main memory or on paging device

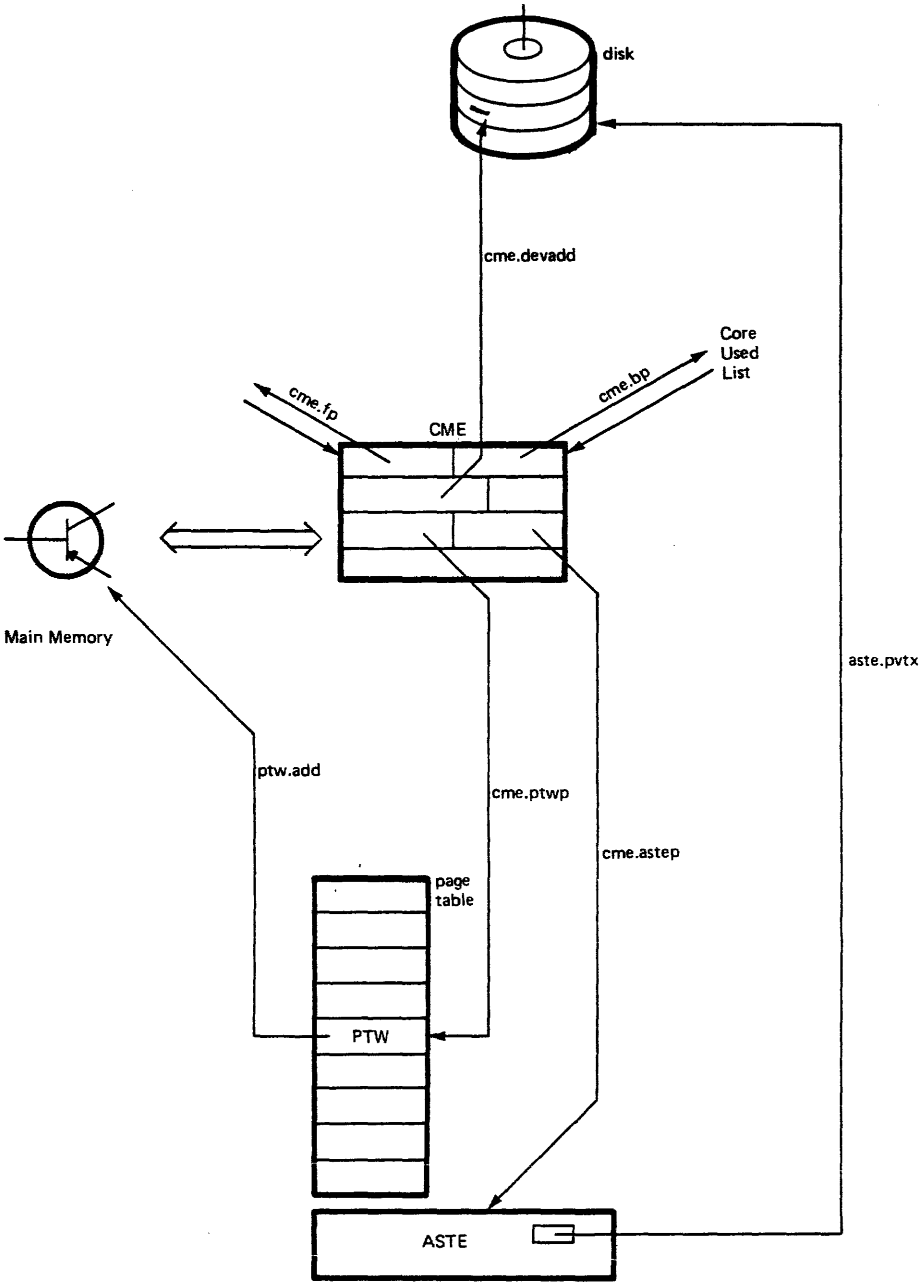


Figure 6-2. Page Control Data Bases
Page in main memory, not on paging device

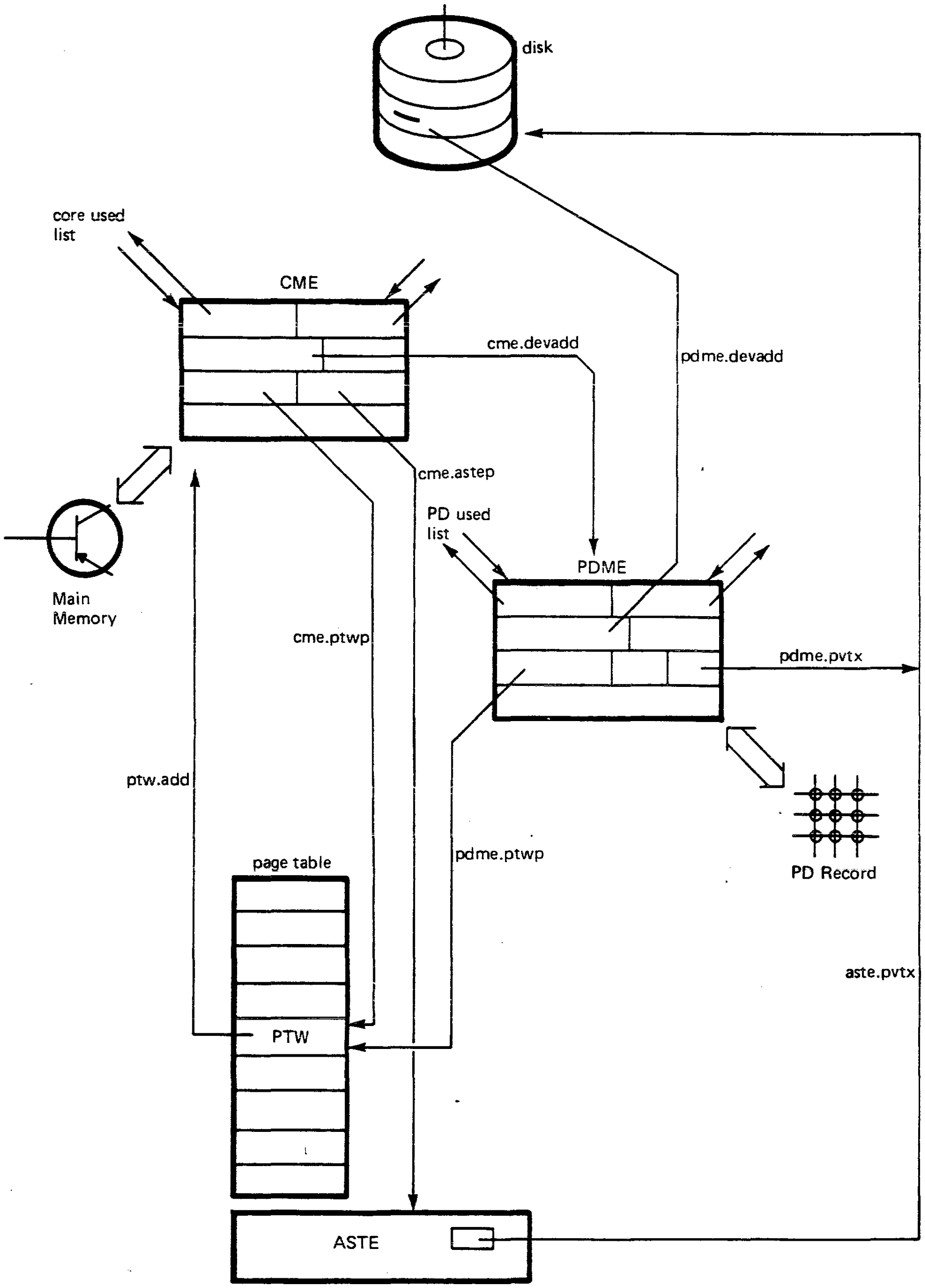


Figure 6-3. Page Control Data Bases
Page in main memory and on paging device

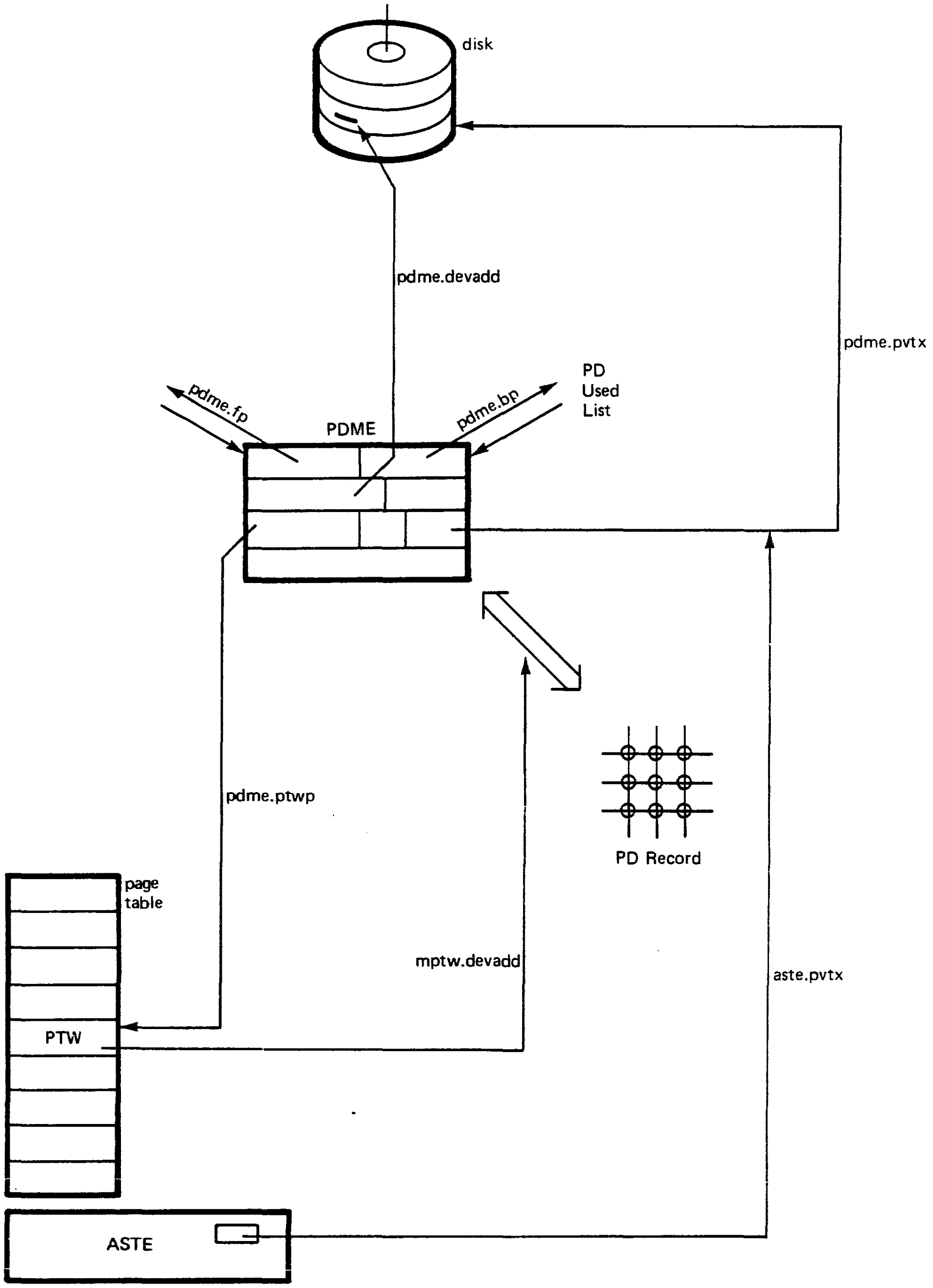


Figure 6-4. Page Control Data Bases
Page on paging device, not in main memory

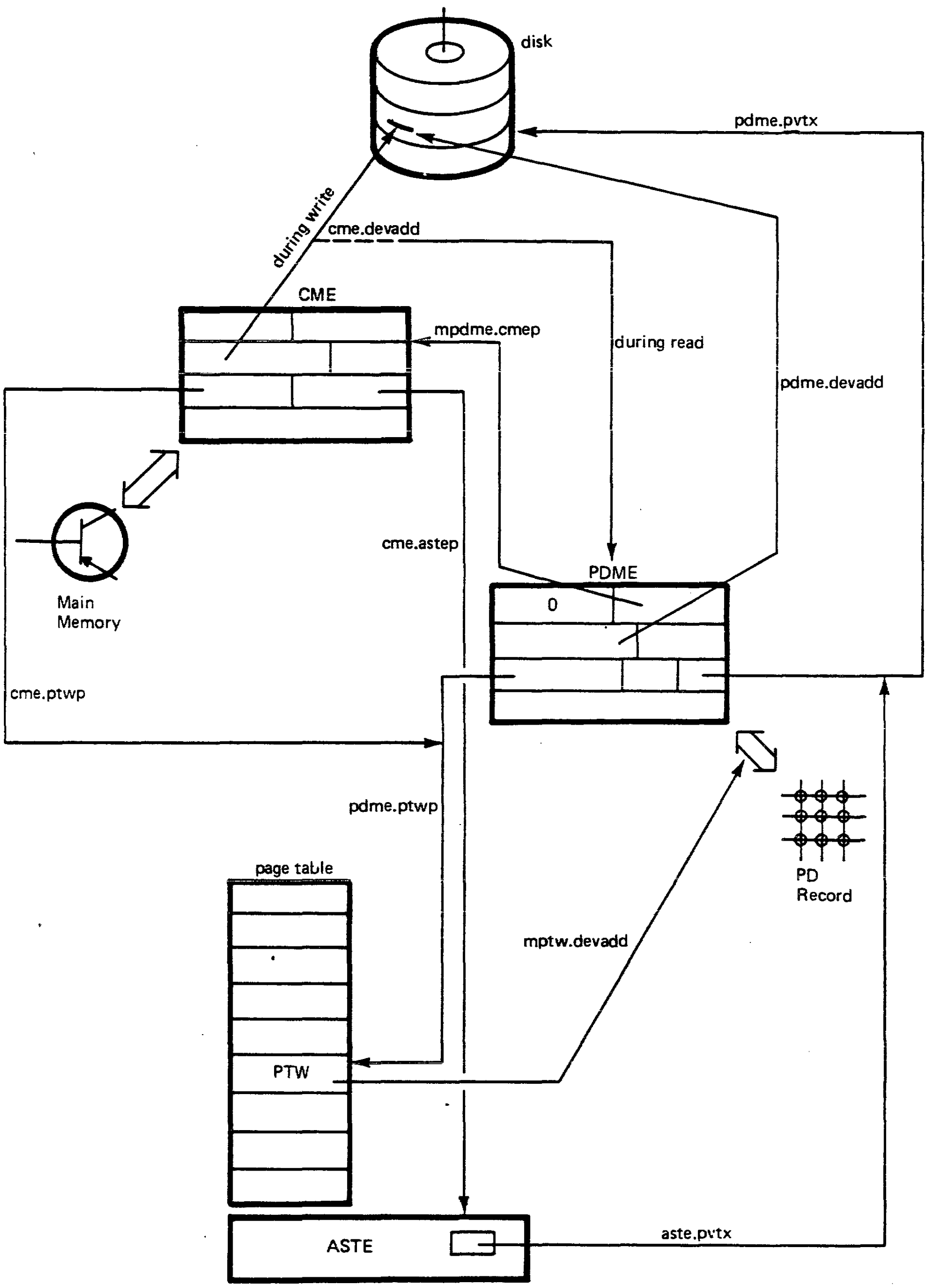


Figure 6-5. Page Control Data Bases: Read-Write Sequence

SECTION VII

ADDRESS MANAGEMENT POLICY

INTRODUCTION AND NULLED ADDRESS

The address management policy of Multics is that set of designs and their implementations which manage when record addresses are assigned to pages, the state of the relationship between the contents of each page and the contents of any secondary storage record which may be assigned to it, and the deassignment of secondary storage addresses from pages.

Some address management policy must exist, as this service is a necessary one of page control, a service to its own internal workings. The goals of the Multics address management policies are these:

1. No record address shall ever appear in a VTOCE unless it is known with certainty at the time it is put there that the data in the associated disk record is the data from the page of the segment which has that address as its record address.

2. No record address shall ever be made available, by placing it in the free pool of records on its physical volume, until it is known with certainty at the time it is so made available, that it has been purged from the VTOCE on disk in which it resided.

3. The observance of points 1 and 2 can be shown to imply point 3, to wit, no record address shall ever appear in more than one VTOCE of a given physical volume at the same time, not even during any transitory or inconsistent states. Such states shall not be allowed to exist.

4. No page of data will be allowed to be created unless a disk record is available to be assigned to it at the time it is created (by being faulted in).

5. The supervisor, when running in any process, shall never encounter a condition where a supervisor data base, stack, or procedure, cannot be grown because of lack of space on its physical volume.

6. The system must be capable of being bootloaded without any knowledge of which addresses are available for assignment. These maps can only be constructed by running software to construct them. This software consists of paged segments, and these segments must reside somewhere.

7. The system shall not deplete its available space on any volume simply as a result of being bootloaded, i.e., shut down and brought up repetitively, or just running an extended or arbitrary period of time.

The address management policy takes cognizance of the fact that the system can crash at any time. A total power failure can cause this. When the system has crashed in such a way that the contents of main memory are lost, or in general, emergency shutdown does not succeed, the next bootload must make the best of what is in the storage system hierarchy as it encounters it. Thus, it is one of the highest goals of address management to make sure the the instantaneous state of secondary storage, at any instant, is never such that the next bootload will give away data by accident or place data in the wrong place.

To understand this more fully, an example must be given of address management policy failure in the pre-4.0 storage system. The following scenario is impossible under the current storage system.

1. Segment A contains a PL/I program. Its owner deletes it, freeing its record addresses, but leaving the data in those pages. The directory file map (predecessor of the VTOCE) is freed.

2. Segment B gets created. Someone types a sensitive letter into it. A record of disk gets allocated for a page of this segment, and is written out. It is a page that used to belong to segment A.

3. The directory page which had A's branch has not yet been written out, as this directory is heavily used, and thus not evicted from main memory.

4. The page of the personal letter gets written out.

5. The system crashes unrecoverably.

6. The next bootload finds segment A still there, as the page of the directory containing the branch never got out to disk. What is worse, one page of this PL/I program now contains a page of the personal letter.

This situation is known as a reused address; due to asynchrony in the updating of pages to disk, two segments claim the same record address. What is worse, the data from the new one is in the page that is described by the file map of the old one. It is the principal goal of the release 4.0 and later address management policy to categorically avoid this and a whole class of similar problems.

It can be seen that if points 1, 2, and 3 above are followed rigorously, the scenario above can never happen. These rules serialize the deallocation and reallocation of addresses so that any trace of any given record is completely gone from one segment before it is freed, and thus made available for use in any other segment.

Point 1 specifically, makes it necessary to make finer distinctions between the states of "there is no disk address associated with a page" and "there is a disk address associated with a page". These finer distinctions did not exist in pre-4.0 versions of the storage system. Consider the case of a page of a segment that has never been written to disk. Now surely, one must allocate a record and associate it logically with that page before writing it, so there must be a finite time between those two operations. There is also the entire time during which the request to write is in the disk DIM queues, when it has not yet been written. Consider the case of a request to "Update the VTOCE" of the segment during this time. Should the address be reported to the VTOCE or not? If it is, and the system crashes before the page gets out, then an address appears in a VTOCE which denotes a record of disk with the left-over residue of some other segment, a security problem. If not, then some finer distinction must be made about the nature of assignment to tell when to update addresses and when not.

This is precisely where the concept of the nulled, or semi-killed device address enters. Point 4 above implies the association of record addresses with pages at the time that null pages are faulted into main memory. A null page is one that is in no way associated with any record of disk, and whose contents are logically zero. The association of this disk record with the page is now in that state given in the precious paragraph, where it is known that it does not contain data from that segment, and may not be reported to segment control. An address in this state is called a nulled or semi-killed address. It is a disk address. It is assigned to a page, but the contents of the page are zero, and the contents of the disk record are residue from some other segment, the nulledness of a nulled address is encoded intrinsically in its representation.

The opposite of a nulled address is a live address. A live address may be reported to the VTOCE, via pc$get_file_map, at any time. Its state of being live implies that that record of disk is known to contained data from the page of the segment which has this live disk address as its disk record address.

The act of converting a nulled address into a live address is called resurrection. Since an address being live means that it is known that a given page has been written there, resurrection happens at the successful completion of any of various disk-writing operations, namely:

1. Any page write from main memory to disk.

2. A read-write sequence (RWS) from paging device to disk.

3. A double-write, when the paging device is being used in write-through mode (see sst.double_write in Section VI).

4. A post-crash repatriation RWS. (See Section IX, "Post Crash PD Flush").

Live addresses can also be dynamically nulled, converting them into nulled addresses. This happens in two cases:

1. When the page is destroyed, via truncate, which includes all cases of segment deletion.

2. When the page is discovered to contain zeros (See "Zero Pages" in Section V.)

When a live address is so nulled, again, zeros become logically associated with the page, and the address is not reportable to a file map. In this case, the page of disk contains a residue again, in specific, the residue of an older version of that page of that segment.

The force of the above policies is that addresses in a VTOCE, as described in the introductory sections of this manual, have only two possible meanings:

1. A Null address: This page of this segment logically contains zeros.

2. A Record address: This page of the segment is contained in the disk record designated.

Therefore, at the time that a VTOCE is updated, the many fine divisions of state of the page and its address must be mapped into one of these two states for the file map being updated, depending on what action is intended for the next bootload should the system crash irrecoverably the next instant. Thus, all states involving nulled addresses are reported to the VTOCE as in case 1 above, via the reporting of a null address to the file map. Now the reporting of a null address to a VTOCE where perhaps previously there had been a live address, is the sole precondition, acceptable to point 2 at the beginning of this section, for depositing (freeing) a record. Thus, at the time that a file map is reported to segment control, a list called the deposit list is also reported: it consists of all of the nulled addresses found in the segment, for pages which were not in main memory or on the paging device (in these cases, it would violate point 4 to deposit their addresses). Page control's association between the page and the disk record is broken at this time by placing a null address in the PTW devadd field and reporting it to the file map, the logical contents of the page remain zero, but no page of disk is associated with the segment.

Segment control holds on to this deposit list. It updates the VTOCE, causing the addresses being deposited to be replaced by the null address gotten above. When and only when this VTOCE write has been determined to be successfully completed, are these addresses (the deposit list) handed in to pc$deposit_list to actually be marked as usable by some other segment. The special entry in the VTOC manager, vtoc_man$await_vtoce, exists solely for the purpose of waiting for successful completion of VTOCE I/O for this reason. The same action is taken when freeing a VTOCE is used as a means of invalidating its contents, when addresses are involved. This is also done by the segment mover. See the descriptions of "VTOCE Updating" and "Segment Truncation" for the impact of these policies on segment control.

IMPLICATIONS OF FINITE PACKS

Each disk pack in the current technology has a finite capacity on the order of tens of thousands of Multics records. Each device address used by page control and segment control is relative to some particular pack: thus the size of these various fields limits, and is limited by, the amount of storage available on one pack.

Each segment resides on one and only one pack: this fact is intrinsic to the interpretation of the device addresses designating records on that pack, as they are only meaningful with respect to a pack designated by the PVT index in the ASTE of the segment in whose data bases they are found. (Note, however, that segments can and do migrate automatically between packs: See Section II).

Since all pages of all segments are assumed to be zero until otherwise known, record addresses are not actually assigned until pages are actually used. In older versions of the storage system, address assignment happened when a page was first evicted from main memory, and was found not to be zero. Since all addresses were withdrawn from the same single large pool, this operation could only fail if the entire system were out of disk, i.e., there was not one more record available anywhere. However, since each pack now has its own pool of free storage, the case of a segment not being able to be evicted because there is no place to write it is a serious one. Such a page would tend to become "stuck" in main memory until some (presumedly complex) action would be taken to recover. An arbitrary number of such pages would tie up an arbitrary amount of main memory. What is more, if the system chose to take a brute-force approach to evict the page, it would have to destroy the user's data, with no particular reason or even good method of telling him or her.

Thus, point 4 above is made. No page of data is allowed to be created (implicitly always as zeros) in main memory, which is the only place pages get created, unless a record is available at that time for assignment. Since it will probably have to be written out later, it is better to find out now if no disk is available. The unsatisfied page fault can be used to make the entire segment-moving mechanism handle the problem transparently if this is done. What is more, the nulled address concept precisely expresses the relation between the page of the segment and the record address so assigned at this time. This unsatisfied page fault is also critical to the implementation of the mechanism that allows page faults on the FSDCT to be simulated by the page fault handler.

It is, of course, always possible that the user process might only reference that page, or never store anything into it but zeros. We cannot rely on that. There is a potential here for interaction with access control to ensure this, but this is not exploited at the current time.

NON SEGMENT-MOVABILITY OF THE SUPERVISOR

The supervisor may not run out of physical volume space at any time. That is to say, if it is necessary to create a page of the supervisor's stack, and there is not a single record available on the volume on which it resides, the system is in an unrecoverable situation. Any software which did any action at all would have to run on that stack, and it cannot be used. Thus, all supervisor data bases, in particular, the ring 0 stack (PDS) of each process, must be assigned addresses at the time it is created as a normal segment, before it is used as a ring-0 stack. This implies a cooperation of page control and segment control. (See "PDS and KST Management", in "Services of Segment Control" in Section IV). Addresses are assigned to the PDS of the process being created by touching every page of it. This causes nulled addresses to be assigned. However, since this segment is part of the storage system hierarchy, the periodic VTOCE update of the AST Trickle (See "AST Trickle" in Section II) would tend to deposit these addresses, as the above paragraphs have stated is the fate of nulled addresses at VTOCE update time. In order to suppress this depositing, the AST bit aste.dnzp, which normally suppresses nulling of the addresses, of zero pages, or checking for them, is viewed in conjunction with the bit aste.ehs, the "entry hold switch" making these ASTE's semi-permanently activated, by pc$get_file_map, to suppress reporting and making-null of these nulled addresses.

This action of pre-assigning addresses is called prewithdrawing. All of the supervisor data bases, such as the stack used at shutdown time, the FSDCT, the dirlockt_seg, the lock segment, etc., are all prewithdrawn at the time they are created by Initialization so that the supervisor does not run out of disk in an embarrassing place. There is another reason for prewithdrawing these segments at the time that they are created: it is a consequence of points 6 and 7, which are now discussed.

GUARANTEED BOOTABILITY OF THE SUPERVISOR

The segments that compose the hardcore supervisor, including all data bases, and all parts of all salvagers, must, if paged, have disk addresses assigned. By virtue of the policies given above, these pages, as all other pages managed by page control, must have addresses assigned at the time that they are created.

If the system has crashed without a successful ESD, then the volume map of any volume present during that bootload will not be valid. (The volume map is the disk copy of the FSDCT bit map for that volume, copied into the FSDCT when the volume is accepted and written out when demounted). The supervisor must have some place to allocate its own pages during the next bootload. Since no volume map may be believed, the supervisor must in effect be booted on a volume not present during the last bootload.

Rather than inflict this difficult operational restriction, a "special volume" called the hardcore partition is defined on the root physical volume (RPV) of a given hierarchy. In effect, every time the system is booted, the supervisor is booted "cold" into the pseudo-volume of the hardcore partition. This is to say that the volume map of the hardcore partition is defined to be entirely full of "free" markings for its pages. Therefore, the supervisor may construct the FSDCT bit-map for the hardcore partition out of "ones" for the length of the hardcore partition. The supervisor may thus allocate pages anywhere in the hardcore partition. (Since the bit-map is wholly fabricated, there is in fact no volume map on disk for this region). The location and extent of the hardcore partition are stated in the volume label of the RPV, and are not subject to change during running of Multics (See Section XIV).

It is a corollary of the definition of the hardcore partition as a region totally free upon bootload that all of the contents of pages in that region, of that bootload, will be undefined (as the records are being reused) during the next bootload. Now only two classes of segments will have pages in the hardcore partition: supervisor segments (without branches or VTOCEs) of that bootload, and deciduous segments (essentially supervisor segments with branches and VTOCEs). The non-deciduous supervisor segment will not be accessible during a subsequent bootload; all information about them was contained in their ASTEs, and is gone. The resources consumed by them in the hardcore partition are reused by virtue of the above definition. The deciduous segments, on the other hand, will have pages all over them being reused by new segments. Therefore, deciduous segments can not be used from one bootload to the next; an attempt to activate a deciduous segment of a previous bootload causes a connection failure. When deciduous segments are deleted, by the next bootload, their pages are not deposited; the records in the hardcore partition are reused by the current bootload by virtue of the definition of the hardcore partition.

All supervisor segments, deciduous and otherwise, are totally prewithdrawn against the hardcore partition with very few exceptions- see below). This means that a given hardcore partition must be capable of holding the supervisor in its entirety, or the system will crash with an out-of-physical-volume condition during initialization. Thus, deciduous segments' record addresses are totally in the hardcore partition, and all of their pages become invalid during the next bootload. This property has been likened to the perennial defloration of flora: that is why deciduous segments are so called.

The bit-map of the hardcore partition is used as the only free storage map for the root physical volume, onto which the system is booted, until the middle of collection 2, when the program accept_fs_disk$rpv runs (See Section XIV). If the system crashed in the prior bootload, the physical volume salvager will have been invoked before this point in the bootload to reconstruct the volume map of the RPV, in addition to other functions. Thus, at this point in the bootload, the real volume map of the RPV replaces the map constructed for the hardcore partition. (No addresses in the hardcore partition should ever be deposited after this point). Thus, all requests for new record addresses on the RPV, will cause records to be withdrawn from the real volume map of the RPV.

The fact that the real volume map of the RPV replaces that of the hardcore partition means that any page withdrawn against that map by the supervisor must ultimately be deposited, or the system will run out of disk on the RPV by virtue of continued operation, a situation explicitly disallowed by point 7 at the beginning of this section. Thus, if supervisor data bases grow, i.e., acquire disk records, after the point mentioned above in initialization (the "acceptance of the RPV volume map", the supervisor must, in order to perform a successful shutdown, truncate these data bases and deposit these addresses to keep point 7 true. Not only is this difficult because of the need to differentiate the hardcore-partition addresses from the ones withdrawn against the real RPV volume map, but this systematic self-destruction of the supervisor causes any problem in shutdown to be hard to diagnose, as the supervisor has willfully partly destroyed itself at that time. It is also difficult to organize a supervisor shutdown which proceeds by destroying itself. (In fact, pre-4.0 versions of the supervisor destroyed themself in just this way, and continually had problems in locating every last record that had to be deposited, and doing it in the right order). Thus, the entire supervisor, with the exceptions noted below, is prewithdrawn against the hardcore partition at the time it is created, for this second reason.

There exists a small set of segments, called "delete_at_shutdown" segments that are managed in complete violation of points 5 and 7. These segments are part of the supervisor. They are data segments that are:

1. Large, and may not even be used for their full length.

2. Non-critical were the supervisor to run out of disk on the RPV were these segments to encounter an OOPV condition.

These segments are managed this way simply to avoid having to make the hardcore partition large enough (an issue of a few hundred records) to contain them were they prewithdrawn against it. Thus, these segments are truncated during a successful shutdown, contain both hardcore-partition and real-RPV-volume map addresses, and may encounter out-of-disk conditions.

The bit slte.delete_at_shutdown, set from the MST generator "delete_at_shutdown" keyword makes a segment so. Such segments are kept in the "hardcore" ASTE list, to facilitate the truncation at shutdown time.

RPV PARASITE SEGMENTS

There are some segments, such as the descriptor segments of all processes except the initializer, and the PRDS of all processors other than the Bootload Processor, which reside on the RPV, but do not have VTOCEs or branches. Thus, page creations for these segments withdraw against the real RPV volume map. In the case of a normal shutdown, orderly process destruction and deconfiguration frees these pages, assuring that the system does not run out of disk by virtue of continued operation (point 7). However, in the case of a crash, with or without a successful emergency shutdown, these orderly destructions do not occur, as all of the relevant processes may be in inconsistent states. Since these "RPV parasite" segments have no VTOCEs, the deletion of process directories performed by system answering service startup does not free their pages. Thus, a volume salvage of the root physical volume (so-called "short RPVS") is performed automatically after every crash. This salvage collects all space not described by VTOCEs, making it available for reuse. This includes all space used by RPV parasite segments.

abs-segs (EXPLICIT ADDRESS MANAGEMENT)

Many "segments" in the supervisor are not segments at all, but rather segment numbers, and possible ASTE/page tables, used for addressing main memory, bulk store, or disk. Such "segments" are known as abs_segs. There are two "levels" of abs-seg, the *SDW-level abs-seg* and the *PTW-level abs-seg*. An SDW-level abs-seg is used by placing an SDW describing a region of main memory (as a segment) in a position in the descriptor segment, or an SDW describing a page table (as the page table for a segment). The extent of main memory, or the segment described by the page-table "become" the "segment" whose segment number was that of the position in the descriptor segment into which the SDW was placed.

For a PTW-level abs-seg, the SDW always describes the same page table. The PTWs of this page table are filled in with the disk addresses of a region of disk or bulk store (the PVT index of that drive or the bulk store (see sst.bulk_pvtx in Section VI) is placed in the field aste.pvtx), and all references to that segment "become" references to that extent of disk or bulk store, i.e., the segment number's segment "becomes" that region of disk or bulk store.

If this reminds the astute reader of the method used to access every single segment in the Multics storage system hierarchy, that is because indeed it is. The difference is solely one in orientation. For an abs-seg, the segmentation and paging mechanism, and the implicit services of page control, are being used as a *technique* to read and/or write disk. For a hierarchy segment, segmentation and paging and the implicit services of page control *and* segment control are used to make a collection of disk records "behave" like a segment. There is no physical difference to the two techniques.

SECTION VIII

MECHANISMS

The mechanisms of page control are those policies, protocols, and programs that compose the internal organization, and support the services thereof. This section details those policies, protocols, and programs. Some policies, such as the address management policy, and the main memory and paging device replacement algorithms, are not manifestations of internal organization, but rather artifacts of the services page control is called upon to perform. Such policies have already been explained.

Those policies already described are the externally visible policies. Some of them have become documented in the literature, and thus acquired some measure of fame. Yet it is the policies and mechanisms explained in this section that are little-known, but necessary to the debugging of problems, interpretation of crash dumps, and contemplations of functional or organizational improvements to the whole of page control

The section is divided into three parts:

1. Policies, protocols, and organizations.
2. Individual mechanisms.
3. Internal interfaces.

The first part describes strategies and principles in effect throughout page control, and critical to its external interface. The second describes particular mechanisms, that are ostensibly divorced from the explicit services, such as the method of waiting for page faults, the "recursive" FSDCT paging, etc. The third part describes interfaces that are in effect the services of page control for page control, such as most of the entries to the transfer-vector "page."

POLICIES, PROTOCOLS, AND ORGANIZATIONS

Global Page Lock

All manipulations of page control data bases, with the exceptions noted below, must be performed under the protection of the global page table lock. No process that has the global lock locked may give away or accidentally lose the processor on which it runs. Thus, any process that has the global lock locked must be masked to "sys_level", and have its stack, linkages, and procedures wired, not referencing any non-wired parameters, code, or data bases.

There is no general mechanism for multiprogram-waiting on the page-table lock. Except for processes taking page faults, all attempts to lock the page table lock are performed by looping on it. Internal to ALM page control, this is performed by executing:

```
      tsx7     <page_fault>|[lock_ptl]
   or tsx7     <page_fault>|[lock_ptl_no_lp]
```

depending on whether or not the caller has set up a stack frame. This procedure may be generally accessed as page$lock_ptl from PL/I code, yet this is rarely done (only the loading function, wired_plm, does this), as all other PL/I procedures that lock the global lock also wish to wire their stack frames and mask to sys_level; this compound function, which includes calling page$lock_ptl, is performed by the very common call:

```
         call pmut$lock_ptl (save_mask, save_ptp);
```

The two parameters are used in the corresponding unlock call:

```
         call pmut$unlock_ptl (save_mask, save_ptp)
```

to identify the PTWs wired by the first call, and the old mask. This mask variable has the old wired bits of the PTWs embedded in it, and is intended for use only by pmut$unlock_ptl.

There exist calls to unlock the page table lock, these involve interaction with the traffic controller in order to support the page table lock multiprogramming feature described in the second part of this section. This call is:

```
      tsx7     <page_fault>|[unlock_ptl]
```

in ALM page control, with the transfer vector page$unlock_ptl and pmut$unlock_ptl having the same relation as the corresponding lock entries (pmut, however, does not use page$unlock_ptl, but rather page_fault$pmut_unlock_ptl, a side door to the unlock mechanism which avoids pushing extra stack frames).

The page-fault handler, the fault side of page control, has a mechanism for waiting, via the traffic controller, for the page table lock to unlock. The lock_ptl routine in page_fault takes special action when invoked by the fault side; this mechanism is explained in the second part of this section.

There are two large classes of page control manipulations that may be performed without having the global lock locked:

1. The turning on/off of wired bits of the PTWs of supervisor or semi-permanently activated segments.

2. The construction or destruction of the page tables of inaccessible segments.

In the first case, the bit ptw.wired, used by the main memory replacement algorithm to avoid eviction of a page, may be turned on or off at any time by any process that is keeping track of what it is doing. Page control, operating under the page-table lock, never turns wired bits on or off except in two cases:

1. Loading of processes' critical pages.

2. Abs-wiring of I/O buffers

Thus, processes may turn on "wired" bits of PTWs for segments such as the ring-zero stack (pmut$lock_ptl does just this) without fear that page control might be trying to turn them off. The restrictions on this type of activity is that one must choose the segment with care: its AST entry must not be removable, lest these PTWs vanish while being dealt with, or before having their wired bits turned off. Thus, only supervisor segments and semi-permanently activated segments (including PDSs of other processes than the initializer) are eligible for such treatment. Furthermore, this mechanism is not shareable; unless some external means is used to organize such wiring requests (such as wire_proc, see Section X, or the I/O Buffer Manager iobm, only segments known to be essentially unshared may be so dealt with (limiting this almost exclusively to ring-zero stacks (PDSs). Once wired bits are so turned on, simply touching the page whose PTW was manipulated, bringing it into main memory, will "wire" it, since it now may not be evicted.

Unwiring of pages so wired may be done by simply turning off the wired bits; it was guaranteed by the preconditions of the last paragraph that the PTWs cannot have disappeared, and no other process could have turned off the wired bits, or worse yet, wanted them kept on. This is the method used to "unload" processes, i.e., unwire their critical pages, without the protection of the page table lock. In fact, an extension of this mechanism is used by the I/O buffer manager to turn off the "abs_wired" bit (cme.abs_w) in the core map entry without the protection of the lock, for the definition of abs-wiring is that the page, and hence, the core map entry it is associated with, may not be moved.

The other broad class of manipulations performable without the page table lock locked is that concerning itself with segments that are inaccessible. A segment being activated by definition has no SDWs describing it, and has no pages in main memory or on the paging device. Thus, any manipulations on its PTWs or AST entry can have no effect on any of the data bases of page control, since no CMEs or PDMEs describe these PTWs or ASTE. A segment that has been "finalized" by pc$cleanup (see "Services," Section IX) again has no pages in main memory or on the paging device; since making the segment inaccessible is a precondition for calling pc$cleanup, such a segment is in the same state, and its PTWs may be dealt with as fitting.

There are two smaller classes of manipulations performable without the page table lock being locked:

1. The validation of page control events by the traffic controller.

2. The depositing of addresses.

The traffic controller interacts in a close fashion with page control to perform Process Loading (see "Process Loading" in "Services"). Among the quantities returned by page control to the traffic controller, when this service is performed, is a wait event. The validity of this wait event is verified under the traffic control lock by the traffic controller, under whose lock all notifications must be performed. This validation is performed by checking out-of-service bits, the particular location of which may be inferred from the value of the "wait event" (see "Wait Protocols" below). If these bits are not on, it is a certainty that the event in question has already happened; if it had not, these bits would still be on, regardless of any lock anywhere, and the traffic controller effectively proceeds with the loading operation, which is, in effect a conservative action for the traffic controller. (The worst possible result of such a mistake would be to retry the loading an extra time.) On the other hand, if the bits are on, the traffic controller assumes that the event has not happened. This is not fully correct; it may have happened already, and a new similar event started. If any such event is in progress, a "notify" will be forthcoming if and only if the "notify requested" bit in an appropriate PDME or CME is on. In the case of the legitimate event being waited for, it always is. In this peculiar case above, it may or may not be. The traffic controller assumes, if the out-of-service (or RWS, as appropriate) bits are on, that a notify will be forthcoming, and sets the process being loaded waiting on that event. The worst possible outcome of a mistake (highly unlikely) in this decision would be a 90-second "notify timeout," and retry.

The depositing of addresses, i.e., the marking of bits in FSDCT bit-maps as free is performed outside of the page table lock. Withdrawing is performed under the protection of the page table lock. The latter is necessary, as were there no lock protecting this withdrawing, two processes might "succeed" in withdrawing the same address simultaneously, resulting in not only a "reused address," but an inconsistent FSDCT and PVT. Thus, withdrawing is performed under the lock. Depositing need not be, because no two processes can be trying to deposit the same address at the same time, because there are no reused addresses in the system. Each address appears at most in one place at one time. Furthermore, no process is specifically trying to withdraw any given address. Depositing consists of turning on a bit and incrementing the free-record count, both of which operations can be done without the protection of a lock. If the address being freed was already free ("unprotected address," a cause for crash) it will be free whether or not the lock is locked. If it is not, no other process is trying to free it. One implication of the fact that depositing is not performed under the page table lock is that the depositing procedure (free_store, called only by pc) takes page faults in the normal fashion on the paged, non-wired FSDCT, while other processes are so doing and the "recursive" page fault simulator is accomplishing "withdraws" on perhaps the same pages.

The page table lock is lower in the locking hierarchy than the traffic controller lock. It is lower than any of the locks used by the storage system DIMs to control their data bases, and thus lower than any locks used by the IOM manager.

It is higher than the lock used by the I/O buffer manager, and thus higher than any locks used by the I/O interfacer.

It is a "wired" (per-processor) lock, and thus higher than any non-wired (per-process) lock, such as all directory locks and the AST lock.

Wait Events Used by Page Control

Page control uses two "waiting" type mechanisms:

1. Looping and retrying until some asynchronous event happens; used to wait for the completion of bulk store I/O, the clearing of the page table lock (by other than the fault side), or the dying-down of disk queue traffic ("running the disk DIM").

2. The wait/notify mechanism of the traffic controller.

The first method is used where giving away the processor is impractical or impossible, including several "worst-case" type situations. The wait/notify mechanism of the traffic controller is used to wait for precisely three types of events:

1. The completion of any disk paging I/O, i.e., disk read or writes of pages to and from main memory for any other reason than a read/write sequence (RWS).
2. The completion of read-write sequences (RWSs).
3. The unlocking of the global page table lock, awaited only by the fault side.

There is also the temp-wiring table used by wire_proc, among the peripheral services of page control, but it is far removed from the internal organization of anything else in page control. (See Section X for more on this.)

Each event for which page control waits has a 36-bit "Event ID," as must be true of all events waited for via the traffic controller. Part of the protocols of using the traffic controller wait/notify mechanism is that event IDs need not be unique over the system, and thus notifies can occur spuriously as event IDs clash. However, event IDs generated by page control <u>are</u> unique within page control. Page control, when looking at an event ID it generated can determine with certainty what event is associated with that event ID, and whether or not it has happened. There are three classes of event IDs corresponding to the three types of events above:

1. A binary number in the right-hand half of a word, whose left half is zeros, this number being <u>bigger</u> than the offset in the SST of the first ASTE (the word offset of the pointer sst.astap), is the offset of a <u>PTW</u> in the <u>SST</u>. Such an event ID is associated with the event of the completion of non-RWS disk I/O for that page.
2. A binary number in the right-hand half of a word, whose left half is zeros, <u>smaller</u> than the offset in the SST of the first ASTE (the word portion of the pointer sst.astap), is the offset of a paging device map entry (PDME). Such an event ID is associated with the event of the completion of an RWS for that PD record.
3. The octal constant "160164153152"b3, being the ASCII for "ptlk", is associated with the event of the unlocking of the global page table lock.

A "PTW event" (Case 1) may be tested for having completed by the being-on of the bit ptw.os. A "RWS event" (Case 2) may be tested for completion by the being-on of the bit pdme.rws in the PDME designated by the numerical value of the event ID. These checks <u>must</u> be made under the page table lock, via an organized methodology explained below ("Wait Protocols"). The "PTL event" (Case 3) may be tested for having completed by inspecting the contents of the page table lock, sst.ptl.

PTW events are also used to express the event associated with the completion of non-RWS bulk store I/O. However, these events never leave page control and thus are never waited for via the traffic controller. Page control "waits" for PTW events corresponding to bulk store I/Os by means of calling the bulk store DIM "run" entry until the event has occurred.

Part 1 - Waiting for a given single event - other than the PTL event (Simplex Wait Protocol)

The methodology used in page control to wait for an event is strongly dependent on which side of page control is doing the waiting. For a start, the interrupt side never waits, or has to wait, for any event (unless loop-locking the global lock is considered waiting for an event). Thus, the interrupt side may not run the replacement algorithm, which would "wait" for disk I/O to die down by looping.

One must consider the code of the process-loading function a separate "side" of page control here; it is the only function that acts on behalf of some given process, including causing that process to wait, but is never actually called by that process.

The page control wait mechanism is not used so that page control may wait; rather, it is used so that processes on behalf of whom page control is performing services may be made to wait, when awaiting page control events is necessary to the fulfilling of that service. This is to say, that when the main memory or paging device replacement algorithms start a write or RWS respectively, page control has no need, in general, to wait for its completion. On the other hand, some process that is trying to drive all pages of a segment out of main memory and paging device may well have to wait for the completion of such a write or RWS, whether it had started it or it had already been in progress. Similarly, a process taking a page fault must be made to wait for a disk I/O completion if a disk read was involved in resolving that page fault. Thus, the procedures that implement the services of page control may often have to wait for I/O completions in order to carry out these services as specified; the mechanisms of page control never wait.

The completion of all page control events is detected and determined by page control. No external agencies in the system wait upon or notify page control events. What is more, the "notify" operation for all page control events is performed under the page table lock, usually by the interrupt side of page control. The occurrence of a PTW event consists of the turning off of the PTW out-of-service (I/O in progress) bit. The occurrence of an RWS event consists of the turning off of the PDME RWS (pdme.rws) bit. These events can only happen under the page table lock. Page control does not perform a traffic-control notify every time a PTW event or RWS event occurs. PTW events are notified only if the bit cme.notify requested in the CME of the main memory frame in which the I/O was taking place is on. These notify operations take place in the traffic controller, but under the page table lock. These notify-requested bits are turned on when and only when page control has made the decision that a process must wait for such an event, at such time, the associated notify_requested bit will be turned on (all under the page table lock).

The decision to make a process wait happens in three different ways, depending on whether the decision is performed by the fault side, the call side (other than the loading function), or the loading function. In the first two cases, the process executing the code will be the one that waits; in the third case it will not.

The fault side makes the decision to wait at the end of page fault processing, all under the page table lock. The readin of the page faulted on, if nonnull has already been initiated (see "Services" Section IX, "Page Fault Handling"). The PTW of the page faulted on is inspected. If the PTW indicates that the page has already been read in (or created, in the case of zero pages), the page fault machine conditions, and thus the faulting Control Unit cycle, and thus the instruction, and the program that took the page fault, are restarted (after unlocking the page table lock). If, on the other hand, the PTW indicates that the page has not been (completely) read in, there is waiting to be done. Since this process has the page table lock locked, and notices that the page is not in, it does not matter whether or not the page has actually come in, i.e., the disk data transfer has been performed. The interrupt side, which is the only agency that can turn off that bit ptw.os or pdme.rws, (cause the PTW or RWS event to occur), cannot be invoked until this process releases the page table lock, or itself invokes the interrupt side under the page table lock. In the case where there is waiting to be done, the subroutine read_page, invoked by the page-fault handler, has returned the event ID of the event that must be waited for. If the page being read in is undergoing an RWS, this is an RWS event. Otherwise, it is a PTW event. If the page requires an allocation of a record, and the appropriate page of the FSDCT is not in main memory, it may be an RWS event or a PTW event for a page of the FSDCT (see "FSDCT paging" later on).

The fault side waits for the event so given to it by read_page in the following way:

> If this event is an RWS event, identify the PDME designated by the RWS event, and turn on the abort bit. This causes an RWS abort and a notify of that RWS event at the time the RWS completes. A branch is executed (pxss$page_wait in the traffic controller) to wait for that event and unlock the global lock.
>
> If this event is a PTW event, determine whether it is for a bulk store transfer or a disk transfer. If the devadd in the CME for the page frame denoted by the PTW is a "paging device devadd," it is a bulk store transfer. Otherwise, it is a disk transfer <u>unless</u> the segment is the "pdmap_seg," abs-seg, an abs-seg used to read the bulk store as though it were a disk. Then it is a bulk store transfer. If it is a disk transfer, turn on cme.notify_requested in that CME, and go to pxss$page_wait to wait for the PTW event. This bit will cause a notify of that PTW event when the I/O completes. If this is a bulk store transfer, call the "run" entry of the bulk store DIM, and check whether or not the PTW out-of-service bit has gone off and call the "run" entry of the bulk store DIM in a loop, until this bit has gone off. The "run" entry of the bulk store DIM will interrogate the hardware status of the bulk store, and call the interrupt side of page control, potentially causing the PTW event to occur, as its function. Then restart the machine conditions.

Bulk store transfers are not awaited via the traffic control mechanism because the transfer time of the bulk store is comparable to the overhead time spent going through the traffic controller.

Thus, a process taking a page fault either restarts the machine conditions at the end of a page fault, or goes to the traffic controller to wait for either an RWS event or a PTW event corresponding to a disk I/O. In either case of going to the traffic controller to wait, a bit will have been turned on (pdme.abort or cme.notify_requested) which tells the interrupt side to notify via the traffic controller.

When a process waits on behalf of the fault side of page control, (this includes waiting for the lock, see "Traffic Controller Interface" below) no other information is recorded about the state of that process other than the machine conditions from the page fault that was taken, and the fact that it is indeed waiting on behalf of the fault side of page control. When that event is notified, the traffic controller branches to page_fault$wait_return, which does not lock the page table lock, modify, or even inspect page control data bases in any way, but only restarts the machine conditions of the fault. If indeed the PTW was made to describe main memory as the interrupt side noticed an I/O completion, and the page has not been evicted in the interim, the interrupted machine cycle will be retried and completed. If not, another page fault will be taken, which will again try to lock the page table lock, perhaps retry page allocation because the FSDCT has now been paged in, or re-read the page if it was evicted in the interim between the time the process received the notify and the time it received the processor. The design is not to determine why the process went to wait; the hardware (by not taking a page fault) or the changed state of page control will do that on their own.

The call side (other than the process loading function) makes the decision to wait when it notices some page with I/O going on, or some PD record with an RWS going on, in a way that interferes with the contract of the entry being called. For instance, if the entry pc$cleanup is called to ensure that no pages of a segment are on the paging device in main memory (the caller having made the segment inaccessible), this surely cannot be true if there are pages being transferred into or out of main memory or the paging device; waiting for this I/O to complete is intrinsic in the contract of this entry. Similarly, the truncate function cannot destroy pages on which I/O is being performed, for the interrupt side at the completion of the I/O would have no way of telling what had happened. Leaving some kind of mark to tell it amounts to waiting for the I/O to complete.

The call side waits by calling page$pwait, with the page tables locked, passing the event ID being waited for as a parameter. Ultimately, if page$pwait so decides, this process will be made to wait. The entry page$pwait, also known as the call side wait coordinator, (its code is in the module device_control) has the following contract:

> Given a page control event ID, with the page tables locked, return when the event has occurred, with the page tables locked.

The call side wait coordinator can always decode the event ID, and by looking at a PTW or PDME, determine if the event has happened. This is the first thing it does (sees if ptw.os or pdme.rws, as befits the event, is off), and if the event has occurred, it simply returns with the page table locked, having fulfilled its contract. (It is sometimes the case that page$pwait will be called with the event ID of an event that has already happened; (see "Multiplex Wait Protocol" below.)

If the event of interest has not occurred, page$pwait decides how to wait for it in the same way as the fault side; if a PTW event for either paging device I/O or pdmap_seg, the bulk store DIM "run" entry is called in a loop until the PTW "out-of-service" bit is turned off by the bulk-store DIM's calling the interrupt side. If this is the case, the page table lock is unlocked, and page$pwait returns with it locked, having fulfilled its contract. If the event is an RWS event or a disk PTW event, the bits pdme.notify_requested or cme.notify_requested are turned on as appropriate, and control is transferred to pxss$waitp in the traffic controller. This entry unlocks the page table lock and waits for the event. When the event occurs, pxss$waitp branches to device_control$pwait_return, which relocks the page table lock (<page_fault>|[lock_ptl_no_lp]), and returns to the caller of page$pwait.

It is part of the protocol of using page$pwait that upon its return, the event might have happened, but the page is out of service again, or that it might have been fraudulently notified. All callers of page$pwait use it as part of the multiplex wait strategy outlined below; implicit in this strategy is the knowledge that these callers will retry all their operations again upon return from page$pwait. Thus, fraudulent notifications are not a difficulty. This situation is exactly parallel to that in which the restart of a page fault upon return of the traffic controller when invoked by the fault side simply retries the faulting machine cycle. No guarantee is made that it will succeed. It is the responsibility of the page control service using page$pwait to ensure that at most a finite number of retries will be necessary (see "Page State Transitions" in this section.

It is necessary that the entries used by the traffic controller to wait for page control events on behalf of the fault side and call side (other than process loading) unlock the page table lock after the traffic controller has locked its own lock. This is necessary to prevent a "lost notify" problem. Were the page table lock unlocked before the traffic controller lock were locked, the interrupt side could run in some other process, between this unlocking and this locking in real time, and the event for which the original process is going to wait will occur and be notified. Then the first process will go to the traffic controller to wait for an event that has already occurred. However, since it is necessary to have both the traffic controller and page table locks locked to perform a notify of a page control event, there is no time at which this notify might come through before the process is set waiting and the traffic control lock unlocked.

The process loading function, as stated before, causes some other process to wait than the one in which it is running. The traffic controller has a special mechanism for this, which will be explained under "Services" in Section IX. The upshot of it is as follows; traffic control will call page control to load a process. Since the process loading function cannot wait, it will either return an event ID, or, by returning zero, indicate that the process is successfully loaded. If not successfully loaded, traffic control will set the process being loaded "waiting" on the event ID returned by page control. When this process is notified, it will not be run, since it is not loaded, but rather, traffic control will call page control to load the process. Page control will either return an event ID, or the fact that the process has been successfully loaded, etc., until the process is loaded.

The process loading function calls page$pread (described in part 3 of this section) to read in the process' two critical pages. This entry calls the bulk store control "run" entry in a loop to wait out any bulk store I/O that it starts. Otherwise, this entry returns a PTW event for disk I/O that it starts, or an RWS event if one is in progress on the page. The process-loading program, wired_plm, (which is in bound_tc_wired, unlike all else in page control) sets the CME or PDME notify requested bits for each event so received from page$pread, or any PTW among those for the process critical pages that were already (or still) being read in at this call. Such a wait event is returned to the traffic controller with the assurance that a notify will be performed when that happens (this is actually using a form of the multiplex wait strategy; see that title below).

Since page control unlocks its global lock before traffic control relocks its own lock, when the process-loading function returns to the traffic controller, there is a window for a lost notify (see above). This is particularly likely on three-or-more processor configurations, where a second processor is likely to hold up the acquisition of the traffic controller lock after a third has just acquired the page table lock. There are also some lost-notify windows because the process-loading function is not in a position to apply the multiplex wait protocol properly.

It is certain, however, that if page control indicates that a process has successfully been loaded, then indeed it has. To rectify this, the traffic controller itself "validates" the nonzero event returned by wired_plm, checking the PTW out-of-service or PDME RWS bit indicated by the wait event, as required. If indeed, a notify was lost, the traffic controller puts that process in tne state where yet another pass through wired_plm will be necessary to determine whether or not the process is loaded, and if not, continue the loading.

Part 2 - Multiplex Wait Protocol

As stated before, the call side of page control does not deal with individual pages at its external interface level. Calls are made to process entire segments, or deconfigure extents of main memory or bulk store, etc. All of the call-side page control entries (in PL/I page control) perform services for the rest of the system on selected groups of pages, records, or main memory page frames. Many of these functions, as noted in the above section, must initiate and/or await the completion of I/O on these various entities. The call side wait coordinator, page$pwait, is provided for this purpose.

All of these functions try to achieve a maximal degree of I/O parallelism (simultaneous I/O operations in progress). This is accomplished by processing all pages, records, or frames in the set being iterated over without performing any waiting. During this iteration, all I/Os or RWSs which need be started are started. As each page or record is processed, a check is made to see if an I/O or RWS is in progress for that page, whether or not it was just started. If this pass completes with no I/Os or RWSs found, then all of the pages or records were processed, and there is no waiting to be done, so the particular function being performed has successfully been completed. If, on the other hand, some I/O was found to be in progress, whether or not this loop had started it, the call-side wait coordinator is called with the event ID of the last such operation notified, and upon return, the entire loop retried, until successfully repeated with no I/Os or RWSs found. This technique is summarized by the following "typical" program excerpt (see any program in PL/I page control for real examples):

```
 1     rt:  event = 0;
 2          do  i = 0 to 255;
 3              if ptw (i) meets-some-criterion then;
 4              else do;
 5                    call page$typical (astep, i, tmp_event)
 6                    if tmp_event ^=0 then event = tmp_event;
 7              end;
 8          end;
 9          if event ^=0 then do;
10              call page$pwait (event);
11              go to rt;
12          end;
```

The variable which is here called "event" is most often called "ind." It is often set to -1 to indicate that no code of the form of line 6 above has ever set it. The call on line 5 above performs some manipulation on a page such as starting an I/O, or continuing an eviction, etc. Such entries, all in ALM page control, perform state transitions upon pages, moving them closer and closer to the particular criterion (such as the one on line 3) which the PL/I program is trying to force to be true. Such criteria are: "No page on this PD record" (for PD record deconfiguration) or "Page not in main memory or on paging device" (for deactivation-time service) or "A good copy of the page exists on paging device or disk" (directory-unlock-time flush service, or shutdown-time main memory flush service). Such entries into ALM page control usually return the event ID of any I/O they start and do not complete, (such as page$pread, which starts page reads). A better set yet, such as page$evict and the "typical" entry above, not only return an event ID for any I/O or RWS they start, but for any they find in progress for that page at the time that they are invoked. Most do not. Some (e.g., page$pwrite) never return an event ID.

We use the term "any I/O or RWS" very loosely. Rather than being a generic class of events, any particular PL/I service (and the ALM entries it calls) might be concerned about either one or both, depending completely upon the semantics of what is intended to be accomplished.

The sort of ALM entries described above, which move pages closer to a given condition, all need some kind of prerequisite condition to ensure that no process or operation will be simultaneously trying to counteract the transitions that the ALM entry is performing. For example, the function that abs-wires portions of segments, pc_abs$wire_abs, calls page$wire_abs on each pair of segment page and main memory frame being abs-wired together, until page$wire_abs reports completion. Before ever calling page$wire_abs, however, pc_abs turns on the bit cme.abs_w, (for abs_wired) for each main memory frame in the region. The replacement algorithm will never evict a page from a frame with this bit on. No process can deactivate the segment, for only supervisor or semi-permanently activated segments are eligible for abs-wiring. Similarly, the deactivation-time service, pc$cleanup, has as part of its contract that its caller must have made the segment being processed inaccessible; thus the transitions performed by page$pwrite, called by pc$cleanup, will not be counteracted.

The PL/I loops using the "multiplex wait protocol" choose one event at random, if any have to be waited for, usually the last one encountered, and to retry the entire iteration, for at least the page associated with this event has changed states noticeably, whether or not other pages have changed state (they usually will have). Similarly, the PL/I function could not possibly be complete until that single event has happened, so it is worth waiting for it. Thus, the choice of event for which to wait is completely arbitrary. If, in fact, an earlier event were chosen, but some later call to ALM page control caused the interrupt side to be invoked and cause the occurrence of this event (<u>post</u> the event), the fact that this event is now invalid is of no issue, as the call side wait coordinator would discover this and return immediately, causing the loop to be redone. (No waste occurs in having the loop redone, for indeed, some I/O which was passed as "in progress" will now be finished, by hypothesis).

As stated above, the process-loading function attempts to use the multiplex wait strategy. However, instead of calling the call-side wait coordinator, which it cannot, and branching to its head, it <u>returns</u> an event ID to the traffic controller, expecting to be <u>called</u> at its entry point when that event has happened. The fact that this arrangement is not an adequate substitute for the complete service provided by the wait coordinator is obvious from the fact that events so returned must be revalidated by the traffic controller.

The various states of pages with respect to the ALM entries that cause state transitions, are illustrated in the section "Page State Transitions," along with the names of the ALM entries or the process actions that cause these transitions to occur.

<u>DIM Interface and "Running"</u>

Page control uses the services of two DIMs, or Device Interface Modules, to manage the I/O operations upon the bulk store and the disks. These are bulk_store_control, the bulk store DIM, and disk_control, the disk DIM.

Page control requires that these DIMs present a uniform functional interface. The semantics of this interface are one of the fundamental internal mechanisms of page control. These DIMs are known as the "Storage System DIMs," to differentiate them from printer or card punch DIMs, etc., or from the user-ring disk DIM, rdisk_.

Page control requires the storage system DIMs to have three entries, read, write, and run. The read and write entries are invoked to request the initiation of read and write operations. The run entry is used to request the DIM to interrogate its hardware status, and call the page control interrupt side if any operations have been completed.

The read and write entries are given three parameters; a device record address, a main memory address, and a word of two flags. The disk DIM read and write entries are also given a PVT index to identify the drive to which the device record address is relevant. The device address and main memory address are those to engage in the data transfer. The word of flags contains two flags, called the interrupt and priority flags. The interrupt flag tells the DIM that it is to call the page control interrupt side when the operation is completed. The priority flag may optionally be used by the DIM to sort the requests received by page control into priorities.

The Disk DIM ignores the interrupt flag, always calling the page control interrupt side. The bulk store DIM does not, however. This feature is used to write out the paging device map to the bulk store every second; as this is not really paging I/O (no PTWs or CMEs are involved), page control does not want the interrupt side to be called upon its completion.

The DIM that receives a read or write request may perform that request in any order it chooses with respect to other requests. A storage system DIM is allowed to call the page control interrupt side while processing the call to start a read or write. Specifically, it is allowed to post the completion of the very request that it was called to perform, should this actually happen. This implies that page control, on return from a call to a storage system DIM to start an operation, must be prepared to find that an arbitrary number of actions have been taken by the interrupt side during that call, including the completion of that operation.

The bulk store DIM operates entirely under the page table lock. Except when called by the Interrupt Interceptor (ii) on account of a bulk store interrupt, it is always called with the page table locked. Bulk store interrupts, however, happen only in the case of a bulk store error, in the current DIM, and the DIM itself calls to lock the page tables in each case.

The disk DIM, however, is called with the page table lock locked at some times, such as when called at the entries defined above, but not at others, as when called by the IOM manager to process a disk interrrupt. At these times, the call to the interrupt side of page control (via page$done) locks and unlocks the global lock itself.

Any storage system DIM may call the interrupt side of page control when the DIM has been invoked by an interrupt. When such an interrupt-time call is made, the DIM must itself (or via page$done) lock the page table lock, and unlock it.

The interrupt side of page control is called by the storage system DIMs with two parameters, a main memory address and a status code. The main memory address is used by the interrupt side to locate a core map entry, from which all other information (such as cme.rws, for example) may be derived. The status code indicates the degree of success of the I/O operation. The low bits of the status code indicate to page control the DIMs determination of whether the problem causing the error is an error in the device, the data path to the device, the record of the device, or the page frame of main memory involved in the attempted transfer. Page control uses this for error recovery (see "Error Strategy" below).

A DIM may retry an operation it has been asked to perform any number of times; page control is only interested in the final outcome. What is more, a storage system DIM can write some page to any number of different records or devices, as it sees fit, and when asked to read it back, read it from any (or all) of them. It is guaranteed by page control that all such copies will be "good." If page control detects that the page was modified when an attempt at eviction is next made, none of them are good; if not, they all are. What is more, a storage system DIM can use the main memory frame into which it is being asked to read for any intermediate buffering, diagnostic results, etc., as long as it contains what was asked for when the operation is posted. Page control makes no assumptions about the contents of page frames that are out of service on reads.

If a storage system DIM given a request to perform, finds that it has no queue space, it is allowed to loop internally awaiting the real-time completion of I/O requests on its devices, possibly calling the interrupt side of page control, if that is necessary to free queue space.

A DIM is allowed to perform services for other parts of the system, as the disk DIM does for the VTOC manager, possibly calling the page control interrupt side when so doing. In such cases, this call must be treated like one on behalf of an actual interrupt.

A DIM must provide a "run" entry, called only by page control with the page-table lock locked, which checks the devices being managed for operations that have completed, and calls the interrupt side of page control for any that have. Such an entry must have two properties:

1. It must physically interrogate the hardware status (perhaps stored) of its device; it cannot depend upon actual interrupts having happened to take cognizance of I/O completions.

2. If called in a loop, I/O operations will be posted one by one via calling the interrupt side of page control, until the DIMs queues hold no more uncompleted requests.

For one example of the use of a "run" entry, see the previous section, where the page fault handler calls the run entry of the bulk store DIM until it finds that the I/O on the faulting page has completed.

The RWS initiator (rws_ in pd_util) "runs" all of the DIMs (calls their "run" entries, one by one, for all two of them) in a loop when more than thirty RWSs are awaiting completion. Thus, it is guaranteed that doing this arbitrarily long will cause an arbitrary number to complete.

The paging device replacement algorithm runs the bulk store DIM to make sure that all read cycles are finished before it is exited, and the page table lock potentially unlocked.

The main memory replacement algorithm runs the DIMs in a loop if an excess (currently 30) of uncompleted page-write I/O requests are outstanding. (The tool file_system_meters reports occurrences of this event.)

The traffic controller "polls page control" every 15 seconds, which consists of calling page$run, which locks the page table lock, runs all the DIMS, and unlocks the lock. This, as all run calls and all other calls, may be used by the DIMS to perform timing-out functions and housekeeping.

Other than calling the bulk store run entry as a substitute for traffic control waits, no page control module other than the ALM program device_control ever calls the storage system DIMs directly. Rather, the entries device_control$dev_read, device_control$run and device_control$dev_write are called. These entries, called only from the ALM page control environment (PL/I page control never deals at this low a level), use variables in the ALM page control environment to determine which DIM to call. This is the function, and the origin of the name, of device control. The call-side wait coordinator also resides here, as well as the page control code called as "page$run" which runs the DIMs on behalf of the traffic controller polling code.

ALM Page Control Environment

All of the ALM programs in page control, including the bulk store DIM, share a common environment of register usage, and all share the same stack frame while in the same invocation of page control. That stack frame is laid out in pxss_page_stack.incl. alm. As can be inferred from the name, the traffic controller shares the same stack layout, which is meant to optimize the case where page control calls or transfers to the traffic controller; in this case, the actual stack frame is shared.

Almost all subroutines in the ALM page control environment invoke each other via the TSX7 instruction; there are a set of "small" subroutines that are invoked with a TSX6 instruction. A subroutine is "small," if it calls no other subroutines.

Any subroutine that calls any subroutine except a "small" TSX6 subroutine must do a "savex"; this operation, performed by the "small" TSX6 subroutine of that name saves index register seven in a stack of saved values in the stack frame. This stack is initialized by the routine init_savex. A subroutine that has not done a "savex" returns via TRA 0,7. One that has returns by branching to the code "unsavex," which pops the stack and returns.

All code in ALM page control, other than the bulk store DIM, runs with pointer register 3 set to point to the base of the SST. Any code that exits the ALM page control environment must restore it.

All external entries to the ALM environment, such as the page fault handler, and the entries called by PL/I code (through the transfer vector "page") are responsible for setting up this environment, i.e., initializing the index register save stack and pointer register 3.

Other than these general conventions, there are conventions of dealing with specific data objects. When any ASTE, PDME, PTW, or CME is being dealt with in any way, all routines expect the following index and pointer register assignments to hold:

Object	Register	Symbolic Name
PTW	Index 2 and Pointer Reg 2	.ptw ptw
CME	Index 4	.cme
ASTE	Index 3	.aste
PDME	Index 1	.pdme

The values in the index registers, are all offsets relative to the base of the SST (pointed to by pointer register 3, symbolically "sst"). These symbolic names are used by most code in the ALM environment to reference these registers. Pointer register 3 also has the names "cme" and "ast" and "pdm" to allow references of the following form to be made:

```
          lda     ast|aste.uid,.aste
```

These symbolic register names may be found in the include files page_info.incl.alm and page_regs.incl.alm.

The use of the stack variables in the ALM page control stack frame is not systematized in any way. No person attempting to modify or maintain page control should change any routine to use any variable that it had not previously used unless they are familiar with every single use of that variable in ALM page control. No attempt is made to document the usage conventions of these variables. This can only be learned via extensive experience with ALM page control. The only variables of any general interest are those named "devadd," "coreadd," "did," "errcode," and "inter." The variables "devadd," "coreadd," "did," and "inter," are the record address, PVT index, and Flag word, respectively, passed to the storage system DIMs. Bulk store control, sharing the same stack frame, uses them in place. The variables "coreadd" and "errcode" are used by the interrupt side to receive the descriptions of completed operations. Again, the bulk store DIM uses them in place. It is also worth mentioning the array "arg," which is used by both page control and traffic control to prepare argument lists and descriptors for any external (PL/I) call that must be made from the ALM page control environment.

Error Strategy

By "error," we refer to any of the following three types of abnormal circumstances:

1. Those resulting from user behavior (e.g., record quota overflow).
2. Those resulting from I/O device error.
3. Those resulting from internal software, or processor error.

The first class of error situation can hardly be considered an error situation at all. The only "errors" in this class are physical volume overflow and record quota overflow. Both of these errors are detected on the fault side; supervisor segments are quota-inhibited (aste.nqsw is on) and prewithdrawn, making these classes of problems impossible. Should they occur on a supervisor or semi-permanently active segment, the system software is malfunctioning, and a class 3 error results. Record quota is checked by the fault handler before any quota cells are incremented; availability of physical records is similarly checked by the record allocation function (free_store) before any data bases are modified. Thus, recovery from either of these circumstances involves no "backup." Record quota overflow is signalled by the fault side on the stack on which the faulting process was running. This is done by moving the page fault machine conditions to pds$signal_data, abandoning the masked/wired environment, and transferring control to "signaller," the procedure responsible for effecting such signalling.

This causes the stack history on that stack to be such that a return to the signaller's frame causes the page fault to be retried. (This is the standard fault-signalling, the only difference here from the common case being that a masked, wired environment, with a stack frame on the PRDS (wired stack segment) was abandoned.) Physical volume overflow is handled by the fault side by marking the ASTE of the segment (aste.pack_ovfl) for which a record cannot be allocated, setting a segment fault in the SDW for the segment implicated by the page fault machine conditions, and restarting the fault. This causes a segment fault to be taken. The segment fault handler locates the ASTE, sees the bit, and invokes the segment mover, presumedly resolving the physical volume overflow situation (see "Segment Moving" in Sections II and IV).

The class of errors produced by detected I/O device failure is that one in which page control policy has the greatest effect upon system behavior. Errors are reported by the storage system DIMs (see "DIM Interface," earlier) to the interrupt side of page control. This severely limits the actions that can be taken at that time. Specifically, no operation that involves waiting can be performed. Furthermore, since the interrupt side can be activated by the call side whenever a DIM is invoked, no action that involves allocating main memory or paging device frames is permissible, since that would involve all of this software recursively. This class of errors may be further subdivided into errors in reading and errors in writing.

Errors in reading are simpler to handle, because there is always some process waiting for the completion of that read. Taking whatever action is necessary and notifying an appropriate event will cause that process to retry that read, either via the fault side retry mechanism or the call side multiplex wait protocol. The response to disk read errors is to turn on the bit ptw.er in the relevant PTW, and return the PTW (otherwise) to its original state before the read was started. Subsequent notification of the associated event causes the fault side to retry, notice the bit, signal an error (condition page_fault_error) (via the same fault-side signalling mechanism as is used for record quota overflow), turning off this bit while so doing. The next retry of that page fault causes another attempt to be made at the disk read.

Errors in reading the paging device (on other than RWS read cycles) are much the same. However, the paging device record involved is dynamically deleted by the interrupt side, because of the fact that an error was encountered in reading it. A syserr message accompanies this action. The disk address (possibly nulled) which was in the PDMAP entry (pdme.devadd) replaces the PD record address (nptw.devadd) in the PTW, causing the next retry of this page fault after the one that signals error to obtain the copy of the page on disk (or zeros if the address in the PDME was nulled).

Errors in reading the paging device for the read cycle of an RWS are somewhat like paging device errors above, although a different error message is printed by syserr. The paging device record is dynamically deleted, and the (possibly nulled) disk address in the PDME replaces that in the PTW. Since, by implication, the RWS has been declared over on account of that error, and the data on disk is thus considered implicitly "valid," the main memory frame of the RWS is freed, and there is no write cycle. No resurrection of disk addresses is performed in this case. Errors during RWS on behalf of the post-crash PD flush are discussed in the consideration of that mechanism in Section IX.

Errors on writing are difficult to handle. While the optimal policy would be to allocate a new disk or PD record, this requires manipulation of the paged segment FSDCT, which is impossible at interrupt time. For the case of write errors to the paging device, the solution is simple; the relevant paging device record is deleted, and the (possibly nulled) disk address from the PDME replaces the PD address in the associated core map entry (CME). This has the effect of forcing the replacement algorithm, or the call side, on behalf of whatever agency is trying to see the completion of this writing, to retry writing, accomplishing a write to disk instead. In effect, the page has been migrated off the paging device.

Errors on writing to disk are problematic in the ways stated. The action at this time is to replace the disk address associated with the page with a null address (page_bad_null, see null_addresses.incl.alm), freeing the main memory frame, causing the contents of the page to become zeros. Errors on writing disk on behalf of the write cycle of an RWS are similar; the null address page_bad_pd_null replaces the PD address in the PTW, and hence, ultimately in the file map. No resurrection, clearly, is performed. Again, special action is taken for the post-crash PD flush.

The third class of errors dealt with in page control is that class of errors indicating software malfunction. In every case, it is dangerous or impossible to continue system operation, since further damage and wrongly disclosed data would probably result. Included among such errors are errors found in locking, errors in expected states of data bases, errors in threading, and so-called "re-used addresses" (records marked as free that are known to be in use, or being freed). Such errors can result only from undetected processor or main memory malfunction, or undiscovered bugs. The effect is to crash the system in every case. In PL/I page control, this is accomplished by calling syserr explicitly. In the ALM environment, the routine page_error is responsible for constructing and executing all syserr calls. There are some entries to this routine (including those used by the bulk store DIM) that report specific errors (such as the non-fatal read and write errors, and paging device record deletions discussed earlier). These routines are knowledgeable about stack variables in the ALM environment, and variable information is printed out in their messages. There are also some entries that crash the system with a specific message, such as that which is invoked upon discovery of a reused address. However, the most commonly used entry is that invoked from the routine page_fault_error in the program page_fault. This routine is invoked from the ALM page control environment via a TSX5 to page_fault_error. It crashes the system with the error "fatal page fault error at location xxxx" where xxxx is the address (in bound_page_control) of the TSX5 instruction executed. In every case, this type of crash is the result of software malfunction, possibly induced by undetected hardware failure. (There is also one case of this type of crash induced by detected hardware (processor) failure; that in which no appending unit cycle bits are on in the page fault machine conditions, indicating appending unit failure.)

There is also a "nonfatal page fault error" facility, which is very sparsely used.

Calls to crash the system via the program page_error call the PL/I routine syserr via a standard call, setting up their argument lists in the array "arg" in the page control stack frame. Part of that PL/I call is the storing of all of the index registers and the AQ at location 40 (octal) in the stack frame of the ALM environment; useful information about the data objects invoked in such a crash can always be gleaned from this data.

The crash for a re-used address is peculiar insofar as the code that invokes it turns on the bit pvte.vol_trouble before crashing. This action causes the physical volume whose volume map was involved to be volume-salvaged the next time it is accepted for storage system use.

Other than these errors, there are no possible errors in page control. No call side entries, or entries to ALM page control return a status code of any kind. No nonfatal failure is possible in the current design. However, in some cases, such as RWS failure due to I/O error on behalf of the post-crash PD flush, status information is conveyed back via the live/nulled/null status of the address left in the PDME by the RWS interrupt side. (See the description of this service in Section IX.)

Stack Management and Interface with the Traffic Controller

Page control uses the wait/notify facility of the Multics traffic controller fairly heavily. The conventions for such waiting and notifying have been discussed.

Page control does not notify any event unless some process is waiting for it, in order to avoid the overhead of traffic control. The bits pdme.notify_requested, cme.notify_requested, and pdme.abort fulfill the function of specifying whether or not such notification is to be performed. All notification is done by the interrupt side, in ALM page control (save for one highly esoteric case during boundsfault processing; see Section IX). All waiting is also performed by ALM page control; the primitive page$pwait serves to perform such waiting on behalf of PL/I page control. The mechanism used by process loading to wait has already been discussed.

The interface between page control and traffic control is streamlined to facilitate these operations. Since the traffic controller and ALM page control share the same stack frame layout, with variables in it allocated to each, the interrupt side transfers directly to a special side-door entry to the traffic controller (pxss$page_notify or pxss$rws_notify) to perform all such notifications. The traffic controller returns to the side-door entries to the procedure page_fault (page_fault$notify_return and page_fault$rws_notify_return) after notifying. The event ID to be notified is passed by page control in the cell pds$arg_1. The quantity seen in the listings as being passed in pds$arg_2 is an obsolete remnant of an old device-metering mechanism. The traffic controller operates completely in page control's stack frame in these cases.

The wait interface is more involved. The interface used by the process-loading function is not discussed here; this has already been treated. The traffic control interface for waiting is always invoked by ALM page control via a direct TRA, from either code in the end of the page fault handler, for (invoking pxss$page wait) causing a process to wait on behalf of the fault side, or from page$pwait, the call-side wait coordinator (invoking pxss$waitp). There is also a third entry, pxss$ptl_wait, used explicitly by the fault-side mechanism that allows multiprogramming to wait for the page table lock. Other than this third mechanism, these entries are entered with the page table locked in every case, being unlocked by the traffic controller after its own lock has been unlocked (see "Wait Protocols" earlier, for the reason this is done).

The interface invoked by the fault side, pxss$page_wait, shares a stack frame from the PRDS with the fault side, which invoked it. The fault-side stack frame becomes a traffic controller stack frame, on the PRDS, and is managed by the traffic controller from that point on as a traffic controller PRDS stack frame, as it is passed around from process to process. Entry to the traffic controller via pxss$page_wait implies that the entire state of the invoking process is encoded in the page fault machine conditions in pds$page_fault_data in that process; this is to say that there is no page control stack history of any kind in that process. Thus, when a process waiting via this mechanism is notified, and subsequently allowed to run, the traffic controller transfers to page_fault$wait_return, which does nothing more than restart those machine conditions (including process/processor mask). Specifically, the page table lock is not locked, nor are any page control data bases at all inspected or modified in any way. This causes the faulting machine cycle to be restarted, either completing successfully (if the page fault has been resolved) or taking another page fault.

When the traffic controller is invoked to wait on behalf of the call-side wait coordinator, a transfer to the entry pxss$waitp is effected. Again, pds$arg_1 contains the event on which it is desired to wait, and the page table lock is locked, to be unlocked by the traffic controller. When a process waits via this mechanism, PL/I page control has a stack history on the PDS of the waiting process; the stack frame that was the current stack frame of that process contains the return pointer to the place in the PL/I program that called page$pwait; that point must be returned to when the waiting has been finished. There are no machine conditions; action upon return from the traffic controller consists of transferring to that place in the PL/I program. Thus, the traffic controller, upon completion of such waiting, transfers to the side-door into the wait coordinator, device_control$pwait_return. Since the page table lock has been unlocked, this entry relocks it via a call to the ALM page control locking interface (page_fault$lock_ptl_no_lp), and returns to the PL/I program at the instruction after the call to the wait coordinator. In order for this policy to succeed, the stack frame pointer register (Pointer Register 6) must be restarted at the time device_control$pwait gains control, to its value at the time that pxss$waitp gained control. Therefore, the traffic controller saves this value in the cell pds$last_sp, which is often useful in debugging problems in this area.

The traffic controller differentiates between the two cases above (fault side wait, no stack history, and call side wait, PL/I PDS stack history) via the variable pds$pc_call, zero for the first case and a positive nonzero number for the second. The value of this variable tells it whether the state of a process waiting for a page control event is embedded in the machine conditions in pds$page_fault_data, or in its PDS stack history, as defined by the value of pds$last_sp. This implicitly tells it whether it should transfer to page_fault$wait_return or device_control$pwait_return.

The mechanism used to wait for the page table lock on the fault side uses exactly the same mechanism as used by the fault side to wait for other events. A special entry to the traffic controller is used in this case (pxss$ptl_wait), which performs certain manipulations as described under "Page Table Lock Waiting" later in this section. However, this special code soon transfers to the code used by the fault-side to wait for all other events. Thus, it is to be noted that the action performed upon notification of the page table lock event is simply to retry the page fault, just like any other fault-side wait.

The variables, pds$last_sp and pds$pc_call, are used by the traffic controller for other mechanisms than page control waiting. Specifically, pds$last_sp is used for all calls to the traffic controller for waiting (other than those just described). The cell pds$pc_call is also used by the traffic controller's preinitialization and shutdown wait mechanism (pi_wait) to differentiate other wait calls than page control's from the two kinds of page control wait already discussed; in this case, pds$pc_call is set to a negative value.

See Figure 8-1 for a synopsis of this mechanism.

In all cases of invocation by ALM page control, the traffic controller is aware that the process/processor are masked to "sys_level," and all relevant parameters are in wired storage. Thus, the traffic controller never pushes its "extra" PDS frame in these cases, because it is used only to store old masks.

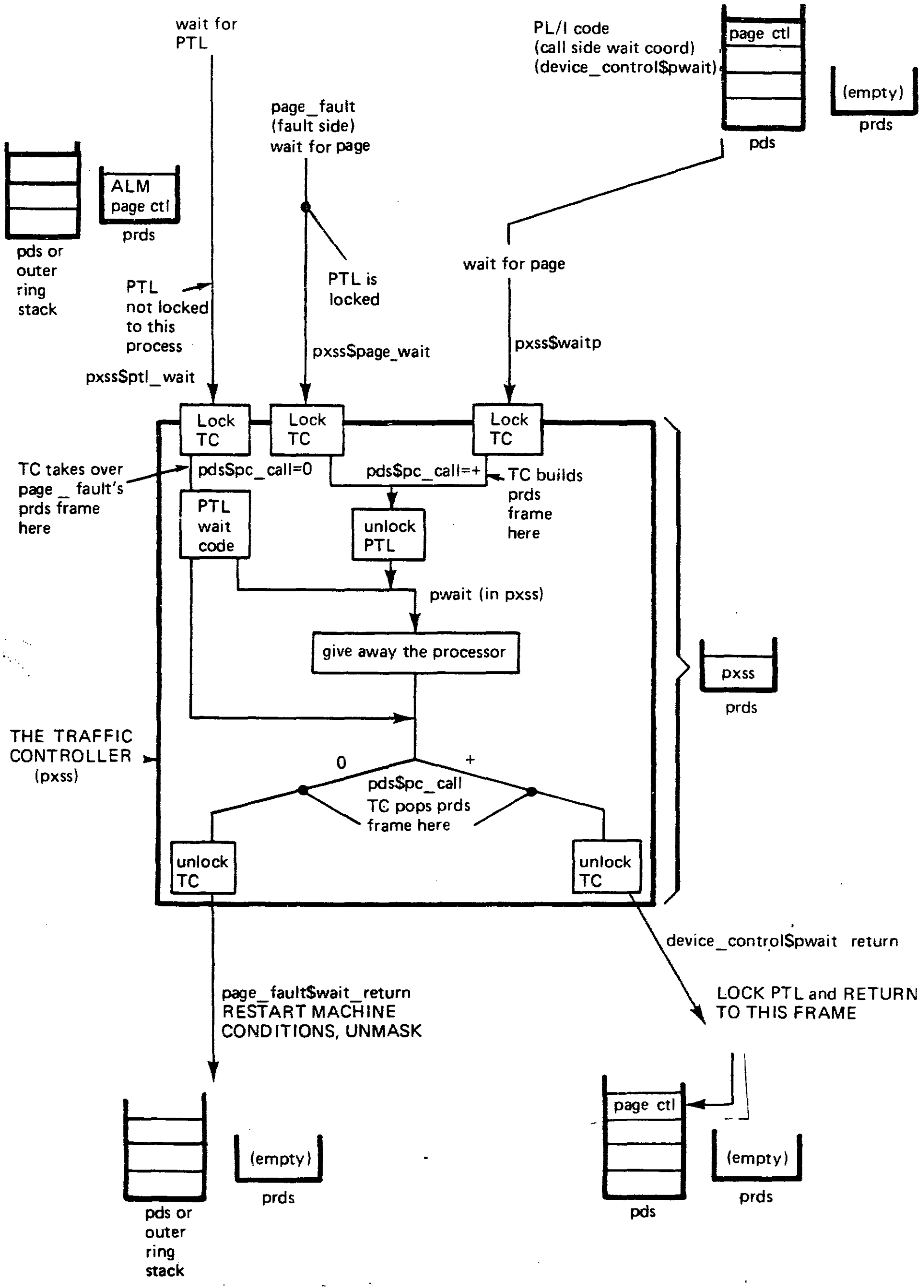


Figure 8-1. Traffic Controller Interface Stack Management

The external entry to page control to lock the page table lock does not need a stack frame; it does not push one (page$lock_ptl, using lock_ptl_no_lp in page_fault). The external entry to unlock the page table lock, however, does, because the traffic controller may be invoked to notify the page table lock event. It pushes its frame, and does a full return (page$unlock_ptl, invoking unlock_ptl, in page_fault). A special side-door is used by privileged_mode_ut$unlock_ptl, however, to avoid pushing a frame. This side-entry page_fault$pmut_lock_ptl, pushes a frame, and explicitly pops it in line before transferring privileged_mode_ut$unwire_unmask to finish the job.

Note that all side doors to page control go directly to individual ALM programs, and not through the transfer-vector "page."

Page States

One instructive perception of page control is that of a set of finite-state automata; one for each page, one for each main memory frame, one for each paging device record, and one for each secondary storage record. The basic operations of page control, specifically the actions performed by ALM page control, consist of performing state transitions upon these objects. PL/I page control, via iteration and the multiplex wait protocol, effects many state transitions at once.

A series of diagrams (Figures 8-1 to 8-7) presenting the various states of these automata is presented here. The entry points, code sequences, or actions that affect each transition are identified. The flags and fields that define each state are identified.

In almost all cases, state transition is performed under the global page table lock. Almost all states of these pages and records are valid when the page table lock is not locked. A notable exception is the read cycle of a paging device read-write sequence (RWS) that is only seen under the protection of the page table lock.

Refer to the discussions of the page table lock strategy and the multiplex wait protocol for more illumination on the motivations for these sequences.

Special mention must be made of the illustrations, Figures 8-2 and 8-3 which show the state transitions of the page of a segment. To avoid over-complication of the diagram, transitions involving paging device deconfiguration (manual and automatic) have been omitted. Also omitted are the data base bit states that denote these page-states as well as program names; again, to avoid over-complication of the diagram. The state of all bits may be inferred from the three previous diagrams. All of the transitions marked "modification" in these diagrams represent not the action of any sequence of code, but rather, that of the user of a given page, in modifying the contents of that page. The transitions and states relating to the use of a page, only of interest to the main memory replacement algorithm, have not been shown in Figures 8-2 and 8-3.

Figure 8-2 is divided into two regions, states of a page that have no associated paging device record, and those which do. The later region is further divided into two regions, those in which the paging device page copy is identical to the disk copy ("PD Notmod") and those in which it differs ("PD Mod").

Two recurring patterns can be seen in each of these regions, the "read-evict" cycle, in which a page is paged in from main memory, used without modification, and evicted; and the "write-mod" cycle, in which a page in main memory is modified by use, written out to "purify" it, and brought thus back to the in-main-memory state of the "read-evict" cycle. When such cycles are isolated, Figure 8-3 is the result, showing the states of a page with respect to main memory and the paging device in terms of these cycles. An entire set of such cycles is shown in Figure 8-4, including the "used" states of interest to the main memory page replacement algorithm.

It should be noted that there are no states in any of these diagrams corresponding to the sematics of the bits cme.abs_wired, ptw.wired, pdme.removing, cme.removing, and pdme.flushing. These bits do not represent _states per se_, but rather instructions to all of the routines that perform the various state transitions as to a desired "goal state." For instance, the flag ptw.wired inhibits eviction, i.e., transition out of those page states where the page is in main memory. The flag cme.abs_wired not only prevents eviction via the replacement algorithm, but any subsequent assignment of the main memory frame to any use (transitions out of the "free" state in Figure 8-5) by any code except that of the abs-wiring function. Thus, the PTW "wired" bit is turned on at any time (see earlier discussions of the page table lock), with the knowledge that any subsequent read-in of the page will cause it move to the "in-main-memory" state and stay there. For the case of wiring a page, the transition to the in-main-memory state is easy to force simply by touching (i.e., faulting upon) the page. In other cases, such as demand eviction on behalf of memory deconfiguration, this is substantially more complicated. Thus, primitives such as evict_page (in ALM page control) exist which, given the appropriate data objects (in this case, a core map entry representing a main memory frame), with such bits _already turned on_, perform whatever transitions are necessary to achieve the desired state (in this case "free"). If the transition is to or from a state where I/O is performed, a wait event ID is returned, otherwise the complete transition is made, and no event ID is returned. The greater part of call-side services such as the abs-wiring and main memory deconfiguration services is to turn on such bits, and call such primitives on each subject page and/or main memory frame repeatedly, multiplexing indicated waits via the multiplex wait protocol.

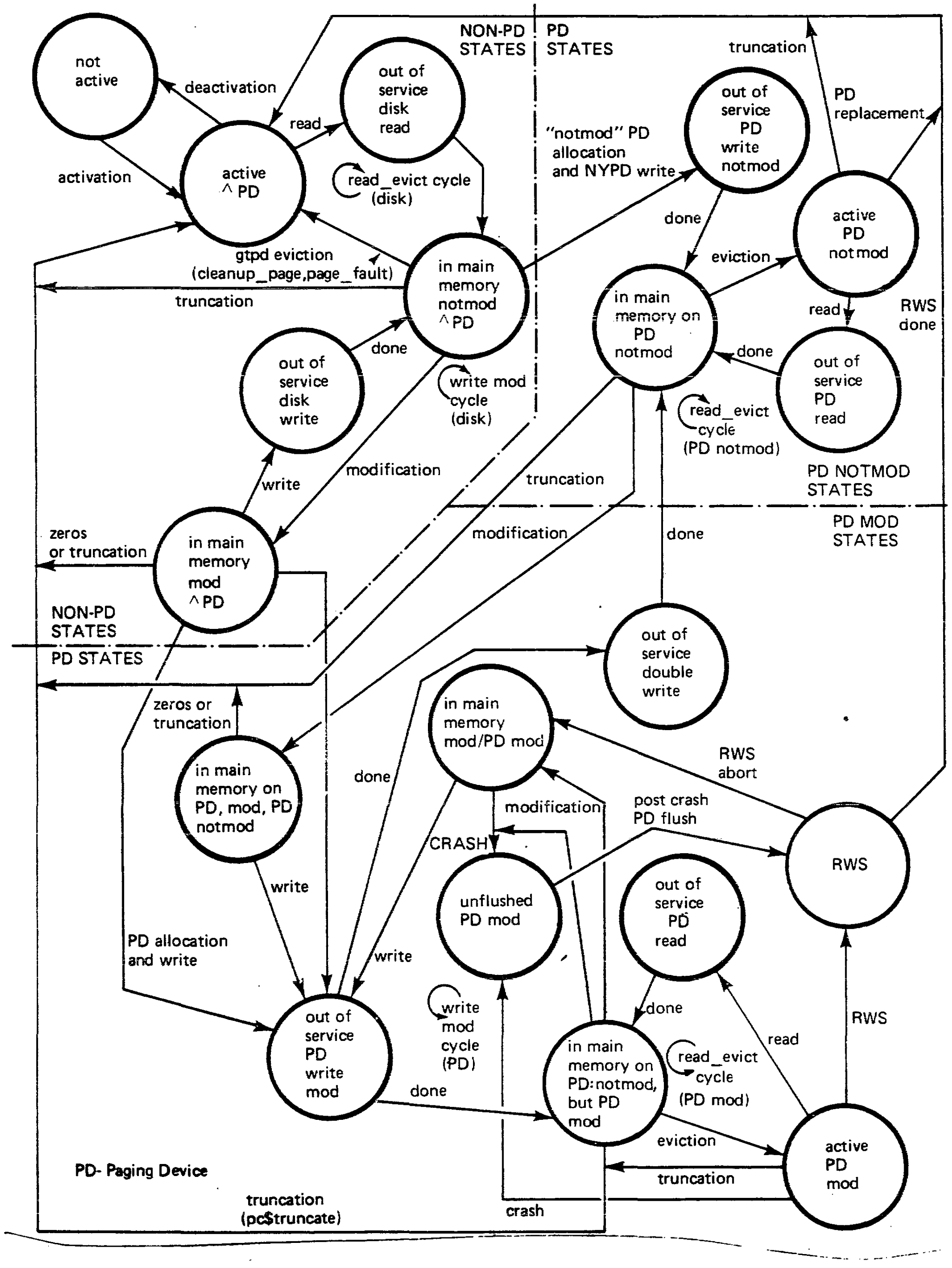


Figure 8-2. States of Page

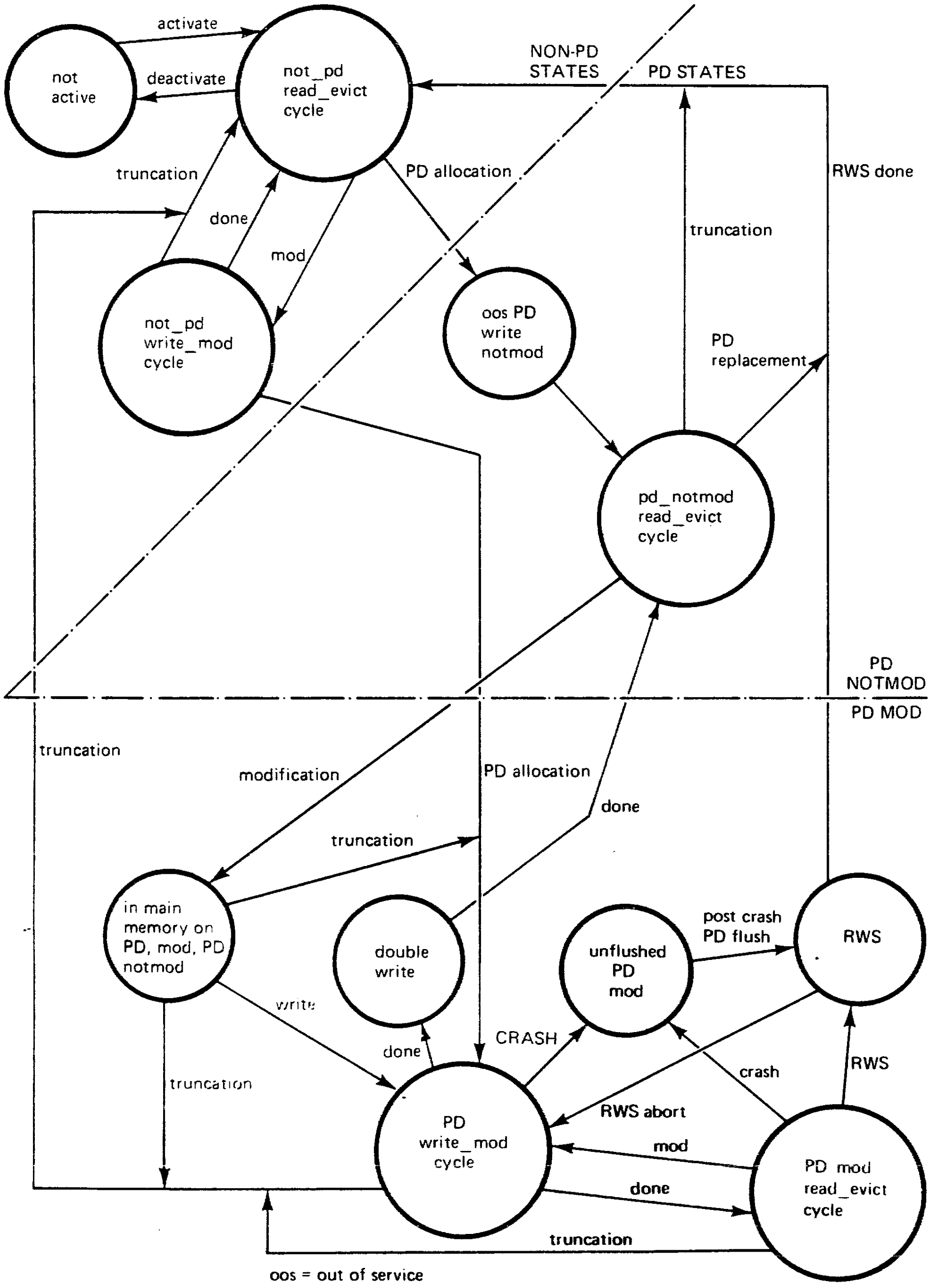
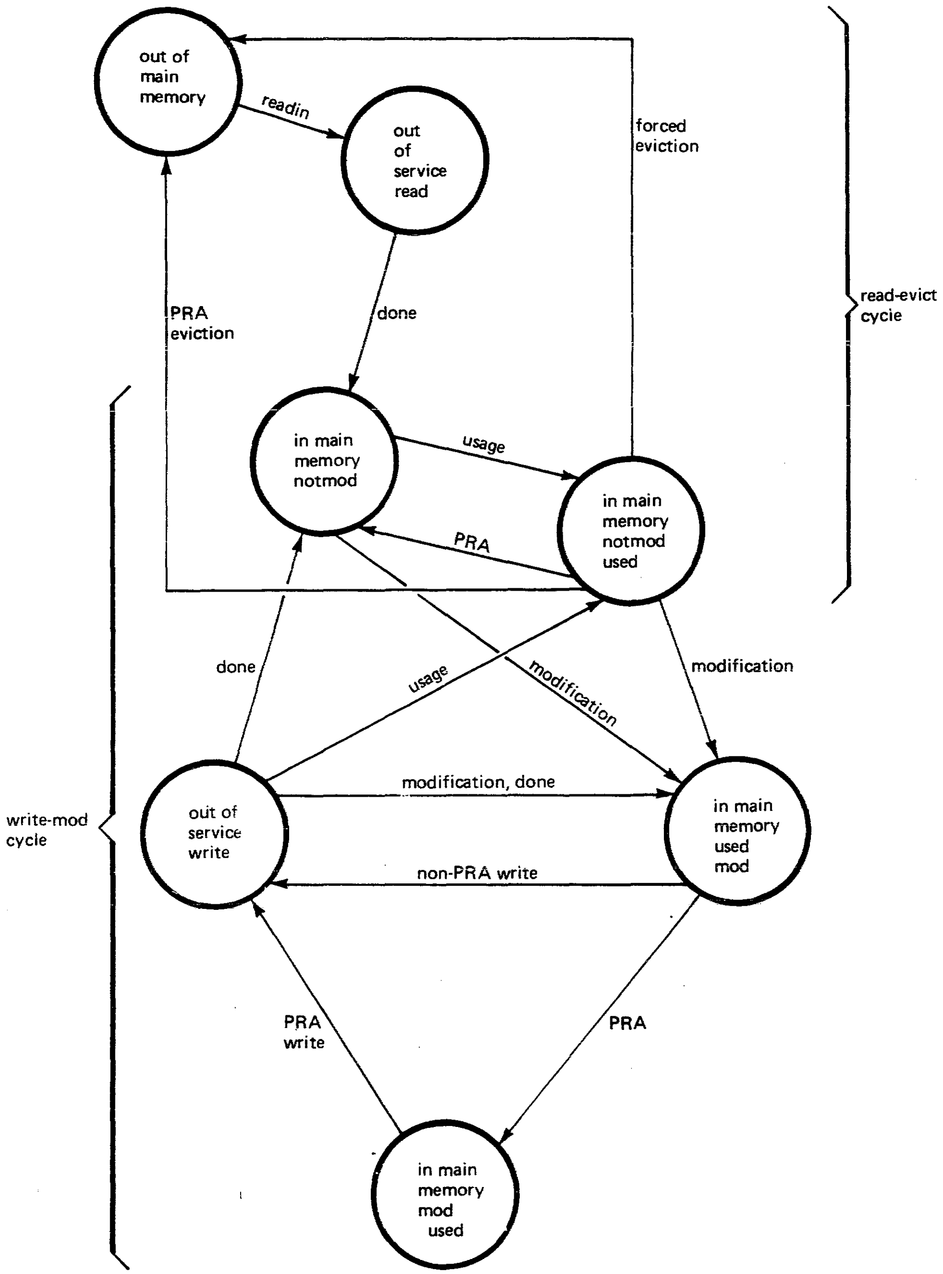


Figure 8-3. States of Page in Macro States



PRA=main memory page replacement algorithm

Figure 8-4. Read-Evict, Write-Mod Cycles

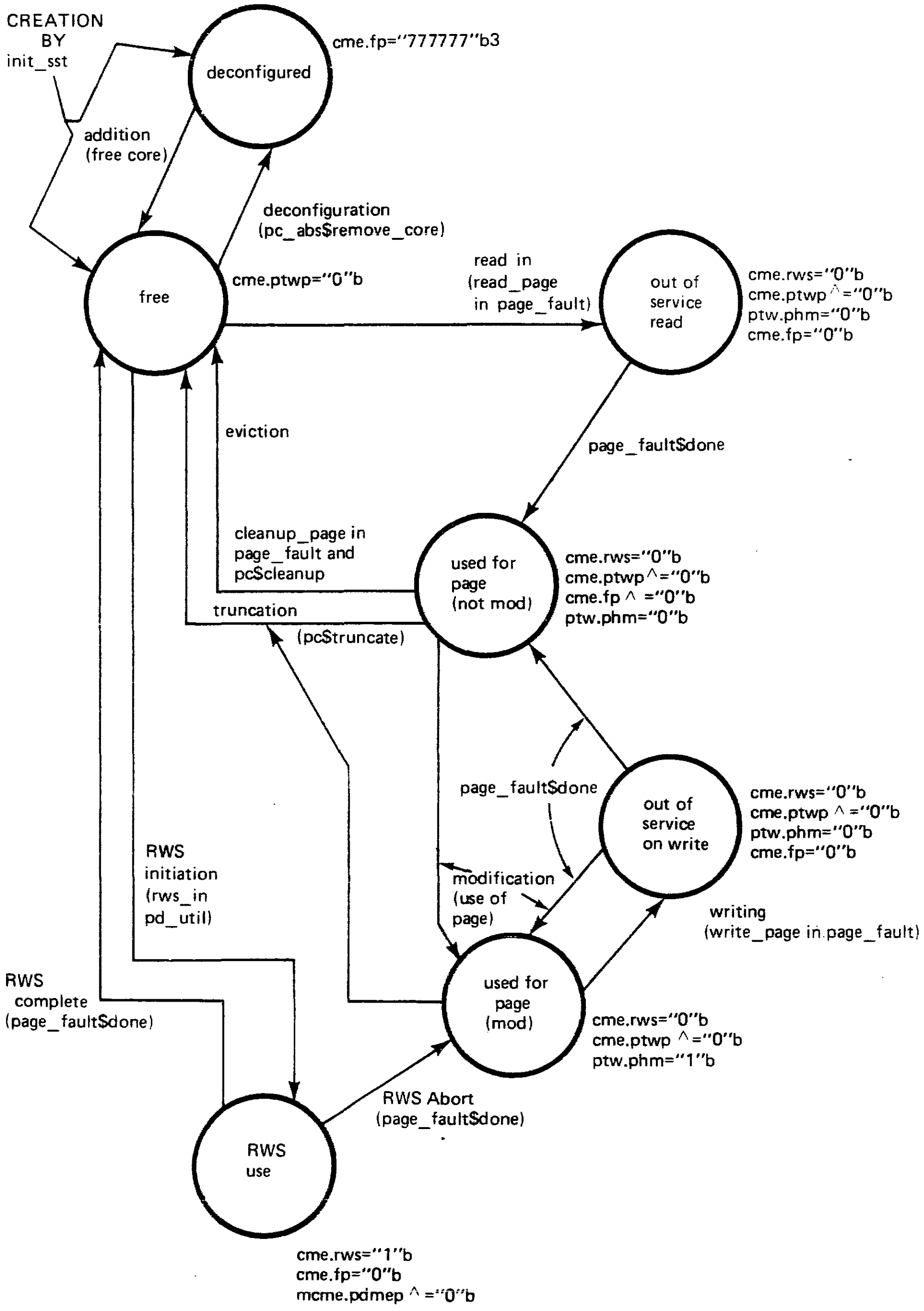


Figure 8-5. States of Main Memory Frames

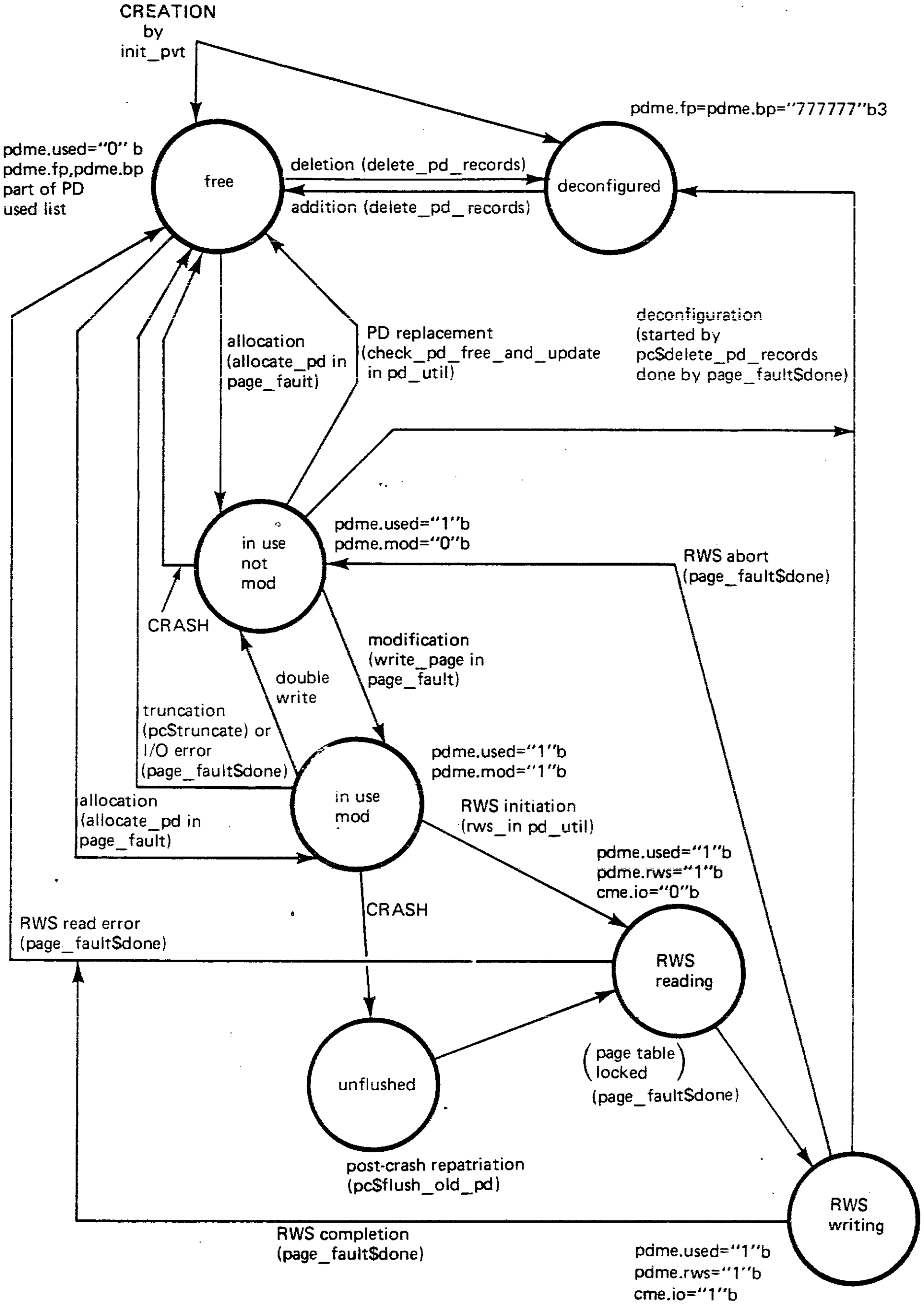


Figure 8-6. States of Paging Device Record

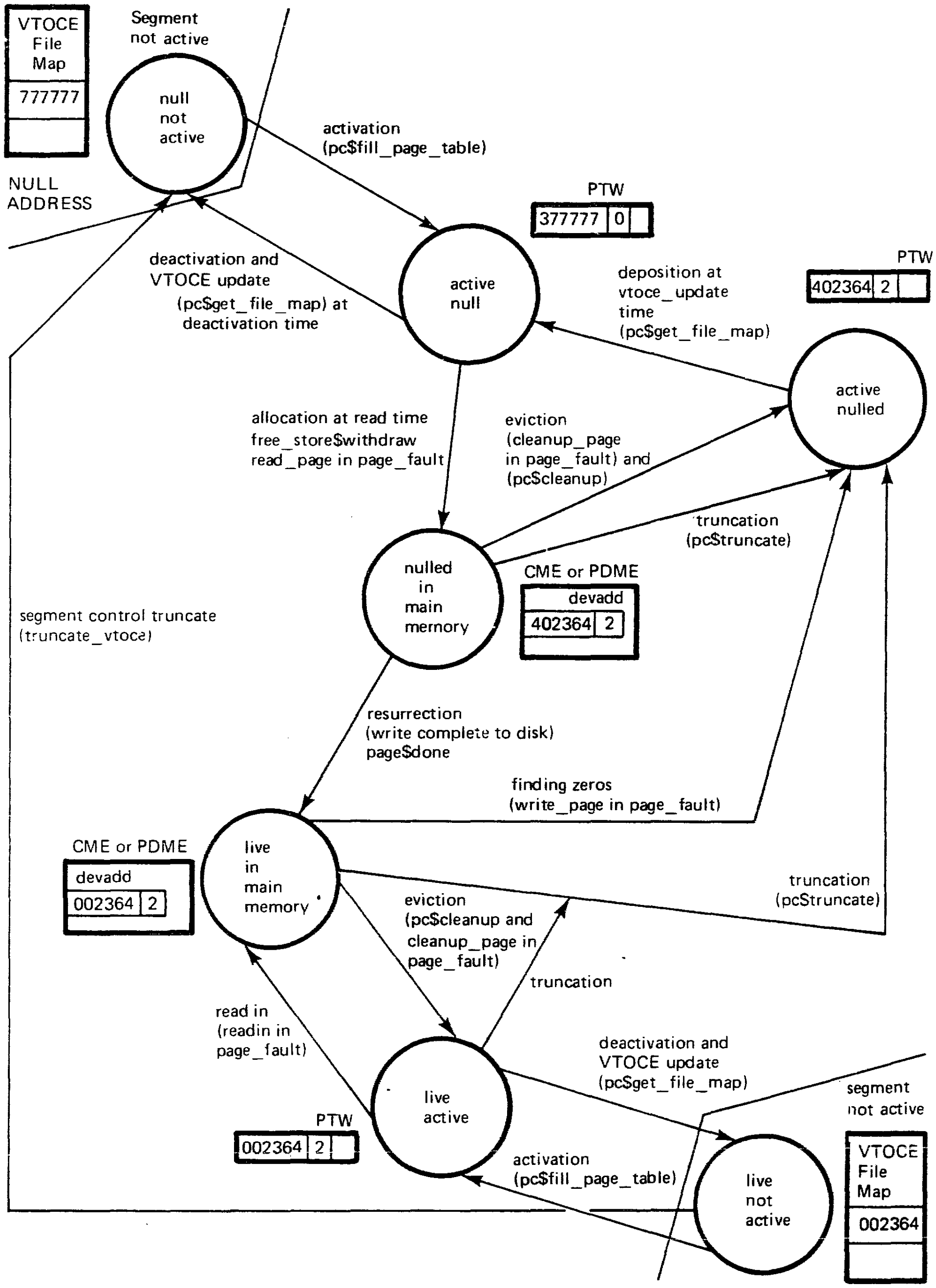

VTOCE File Map
777777
Segment not active
null not active
NULL ADDRESS
activation (pc$fill_page_table)
PTW
377777 0
active null
deactivation and VTOCE update (pc$get_file_map) at deactivation time
deposition at vtoce_update time (pc$get_file_map)
PTW
402364 2
active nulled
allocation at read time free_store$withdraw read_page in page_fault
eviction (cleanup_page in page_fault) and (pc$cleanup)
truncation (pc$truncate)
nulled in main memory
CME or PDME
devadd
402364 2
segment control truncate (truncate_vtoce)
resurrection (write complete to disk) page$done
finding zeros (write_page in page_fault)
CME or PDME
devadd
002364 2
live in main memory
eviction (pc$cleanup and cleanup_page in page_fault)
truncation (pc$truncate)
truncation
read in (readin in page_fault)
live active
PTW
002364 2
deactivation and VTOCE update (pc$get_file_map)
segment not active
activation (pc$fill_page_table)
live not active
VTOCE File Map
002364

Figure 8-7. States of Disk Address

Tracing Mechanisms

There are two tracing mechanisms in page control, both of which have not been maintained in recent years.

The "page control trace" mechanism is part of the hardcore system trace facility. It is enabled by switch 34 on the Processor Maintenance Panel switch register. This switch register is read and stored in sst.trace_sw on every page fault. When enabled, this trace facility causes tracing messages to be printed or written to tape, as selected by the system trace facility Various callside routines (mainly to the program "pc") inspect this switch, and call "trace," the system trace routine, with arguments describing the action being performed, and the location and contents of the AST entry upon which they are being called to operate. Many actions in ALM page control are traced as well; they can be located via the calls to "pc_trace" in ALM page control.

The program pc_trace is part of ALM page control. It is invoked at its various entries, each of which traces one type of event, via a TSX7 instruction from within bound_page_control. This program issues no messages; rather, it sets up argument lists for the program pc_trace_pl1 which does. These argument lists are functions of the individual entries. The actual arguments are particular stack variables and index register values from the invoking ALM page control environment. The program pc_trace_pl1 contains nothing but the PL/I calls to the system trace facility, referencing the arguments passed by pc_trace.

The second trace facility in page control is that referred to internally as "disk_meters." This facility is the remains of an experiment described in the MVT Project MAC Technical Report cited in Section V, which accumulated traces of paging device allocations and evictions in order to achieve performance predictions for extended paging devices and main memories. This facility is enabled and disabled via the program "get_disk_meters," which wires and unwires the tracking buffer, "disk_traffic_data." The trace entries are accumulated by the program "meter_disk," invoked from ALM page control via a TSX0 instruction. All entries to this procedure start with an XEC instruction; when this facility is not enabled, the target of this instruction is a TRA 0,0, which returns at minimal cost. This facility has not been functional since release 4.0; furthermore, there are no installed tools to retrieve or interpret its output.

INDIVIDUAL MECHANISMS

Waiting for the Page Table Lock

The fault side of page control has the ability to utilize the traffic-controller wait/notify mechanism to wait for the page table lock to be unlocked. This ability depends upon the fact that the fault side has not modified any data bases or changed the state of its process at the time that it encounters the page table lock locked. Thus, if that process is made to wait for the unlocking of the lock, via the traffic controller, the return from that wait may simply restart the machine conditions of the page fault, probably taking the page fault over again and retrying the operation. Thus, it may be seen in Figure 8-1 "Traffic Controller Interface Stack Management" that the entry to the traffic controller to await a page from the fault side (pxss$page_wait) ultimately merges with that which awaits the page table lock's unlocking (pxss$ptl_wait), both returning to page_fault$wait_return to restart the fault.

Since processes may be waiting for the unlocking of the page table lock, it is potentially necessary to notify the "PTL event" (the page table lock event ID, "160164153152"b3) every time the page table lock is unlocked. Since there is a substantial overhead involved in calling the traffic controller notify primitive to do this (it may involve looping on the traffic controller lock), there is a means to avoid this notify call when in fact no process is waiting for the unlocking of the page table lock. This means is implemented by the cell sst.ptl_wait_ct. This cell is zeroed only by the notify code in the traffic controller when it notifies the PTL event, protected by the traffic controller lock. All code that unlocks the page table lock inspects this cell after unlocking it; if nonzero, it notifies the PTL event.

Any process that finds the page table lock locked on the fault side transfers to pxss$ptl_wait. This entry, once it locks the traffic controller lock, increments the cell sst.ptl_wait_ct. From this point on, any process that unlocks the page table lock must call the traffic controller to notify the PTL event. The page table lock is then inspected, under the protection of the traffic controller lock, to see if it has been unlocked since the fault side found it locked. If so, the process is made to wait for the PTL event, since it is guaranteed that sst.ptl_wait_ct is nonzero (as it is only zeroed under protection of the traffic controller lock, now held by this process), and thus, that the PTL event will be notified, even if the page table lock has been unlocked since the last check, for the process that unlocked it checks afterwards the contents of sst.ptl_wait_ct. On the other hand, if the page table lock is found to be unlocked at this second check, the cell sst.ptl_wait_ct is decremented by one, as this process will not wait, but retry. Thus, in this case, the process returns out of the traffic controller as if the PTL event had been notified, causing the page fault to be retried.

There are two code sequences in the system that unlock the page table lock. One is the subroutine unlock_ptl in page fault, and the other is the code in the traffic controller page control wait entries that unlocks the page table lock once the traffic controller lock is locked. The unlock_ptl subroutine checks sst.ptl_wait_ct and calls the traffic controller page-control event notification routine, via the stack-sharing mechanism described earlier in this section. Normally, this routine is only called from the interrupt side "done" code of page control; this is the point to which the traffic controller returns. The value of the event ID (the PTL event), which in this case can only be notified from the unlock_ptl code, causes the interrupt side routine to return to the unlock_ptl code. The return address, being the value of index register 7, is saved in pds$arg_3 during this call.

The code sequence in the traffic controller that unlocks the page table lock calls an internal traffic controller notification primitive ("n3") to notify the PTL event if sst.ptl_wait_ct was not zero after the page-table lock was unlocked.

FSDCT Paging

The segment FSDCT in ring zero contains all of the free-storage bit maps for all mounted physical volumes. This can grow to be quite large, and thus, this segment is a pageable segment, subject to demand paging behavior for its entire extent. The information in the header of this segment is used by volume management, where the pageability of this segment presents no problems. Similarly, all deposition of addresses (freeing of disk records by turning on bits in this segment) are done by segment control (the update_vtoce and truncate_vtoce functions), and some of the peripheral services of page control (e.g., pc$truncate_deposit_all). Again, pageability is no problem. The withdrawing of addresses, however, is performed by the page-reading function in ALM page control, invoked from the fault side and various functions on the call side (such as abs-wiring for I/O buffer usage).

The ALM page control kernel may not itself take page faults. However, a mechanism exists to allow the page-reading function to achieve paging-in of the FSDCT without taking a page fault. This mechanism relies on the multiplex wait protocol and the fault-side retry mechanism. More precisely, either the fault side or the call side, when made to wait for an event via the traffic controller, will retry the operation that caused them to wait for that event. In the case of the fault side, this means taking the original page fault over again. In the case of the call side, this means re-evaluating PTW states, and calling ALM entries to perform state transitions based upon these decisions. The essence of the mechanism is to initiate the read-in of the needed FSDCT page (when allocation is required) _instead_ of the requested page, and causing the faulting process or the call side to wait for the event associated with _this_ paging-in instead. When this read-in is finished (the event is notified), the page-read function will probably find the needed FSDCT page in main memory, and thus be able to proceed as though it were there to start with.

It is not ensured in any way that the FSDCT page paged in in such a manner will still be in main memory when the page-read function inspects it again. In this case, another read will be started for it, and the operation repeated. This is as deterministic as an ordinary page fault; it is not _necessary_ that this operation complete in any given number of retries, but simply optimal to the behavior of the affected process. Similarly, the disk-record allocation function (free_store$withdraw) may progress through several pages of the FSDCT to find the necessary allocation. This will cause these pages to be paged in successively, with the faulting process or the call side being made to wait for each one in succession. Between the time that these processes are made to wait and the time that they retry the search through the actual FSDCT for a free record, other processes can deposit pages (paging in the FSDCT via normal paging) and withdraw record addresses (via the same mechanism). There is no interlock against this, or any need for one. The state of a given bi-map is recorded in the PVTE for that volume, and is not dependent upon any allocation that might be in progress.

The necessity of the read-page function to have the necessary pages of the FSDCT in main memory to complete its task is very much akin to the necessity of having a set of pages in main memory to initiate execution of multi-operand EIS instructions; nothing ensures that all the required pages will ever come into main memory, although every retry attempt tries to bring them there. Once they are all there, the operation proceeds. The process is effectively roadblocked until all these pages can actually be found in main memory at once; how long this is depends solely on system paging load.

The interface to the disk-record allocation function involves two error exits; one for the out-of-physical volume condition (no more records to allocate) and another for a needed page of FSDCT not being in main memory. In the latter case, the AST entry pointer and PTW pointer for the needed page are returned in the AQ register to the page-read function, which now _redefines_ its task to be the reading-in of _that_ page, including the allocation of a main memory frame and all other actions normally associated with the page-read function (see "Page Reading" later in this section). The last step of this function is to return a wait event to the caller. In this case, it will be that wait event associated with the FSDCT page.

Per-Process Trace List

(page trace)

Page control maintains in the PDS of each process a circular trace buffer of page readings, being mostly page-faults. The primary use of this trace is to drive the post-purge function at eligibility loss time (see "Post Purge" under "Services"). A secondary use is for the user commands "page_trace" and "cumulative_page_trace," which display and interpret this information. To the latter end, various other mechanisms in the system make entries in this trace list corresponding to such events as linkage faults, segment faults, and schedulings. The trace region is at the symbol pds$trace, in the wired part of the PDS. The format of this region is given in the include file page_trace.incl.pl1, as well as the format of the trace entries.

This trace list is maintained by the subroutines "page_util_enter" and "enter" in page_fault.

Disk Record Allocation/Deallocation

A bit map of unallocated records on every physical volume is kept in the segment FSDCT. The parameters that describe each bit map, including the offset of the bit map in the FSDCT itself and the state variables of the allocation/deallocation mechanism for that volume, are kept in the PVT entry for the volume concerned (see "Data Bases," Section VI).

The basic allocation strategy maintains a pointer into each map (pvte.curwd), that points to the last word in the map in which free records were found. Each word of map describes 32 records. When a request is made for a record, that word is scanned for another one-bit. Successfully finding a one-bit on causes the record defined by that bit to be returned (allocated), the bit being then turned off. The result of the floating-point normalization is checked by testing that the bit it claimed to be on is actually on; failure produces a "unprotected or reused address" crash.

Before any word of the FSDCT is inspected, a check is made (by inspecting the FSDCT's page table) that the necessary page of FSDCT is in main memory (see "FSDCT paging" earlier). Before any allocation is attempted, a check is made to see if there are any free records on the specific volume at all; if not, an error return is taken causing the ultimate invocation of the segment mover.

As each word is depleted of free storage bits, the next word in the bit map is moved to. The code that accomplishes this (in "withdraw" in free_store) contains the remains of an algorithm which used to interlace assignment over drives, prior to the advent of physical volumes. The effect of this in every case is to move on word-by-word up the bit map, and come around again to the beginning when the end has been reached. Thus, the pointer pvte.curwd cycles through the bit map for each drive.

Whenever one hundred deposits are made against a given drive, the deposit code resets this counter (pvte.relct) and resets the "curwd" pointer to zero. This has the effect of packing records tighter on each pack; whenever one hundred records have been deposited, the scan for the next free address is thus reset to the lowest address in the paging region of a pack.

The code for depositing (freeing) an address is trivial; the bit corresponding to that address is turned on. If already on, a reused-address error has occurred, indicating page control malfunction, and the system is crashed.

Any reused address detected by the program free_store caused the "vol_trouble" bit in the PVT entry for that volume to be turned on; this causes a volume salvage the next time that volume is accepted, even if ESD succeeds.

INTERNAL INTERFACES

This section explains the structure and function of the basic page-state manipulating subroutines of ALM page control. Some are externally accessible from PL/I page control via the transfer vector "page." Many are not; it is the functions provided by these interfaces in terms of which the Page Control Services of Section IX will be described. Figure 8-8 shows the call flow of most of these routines. Utility subroutines are described in the section following this.

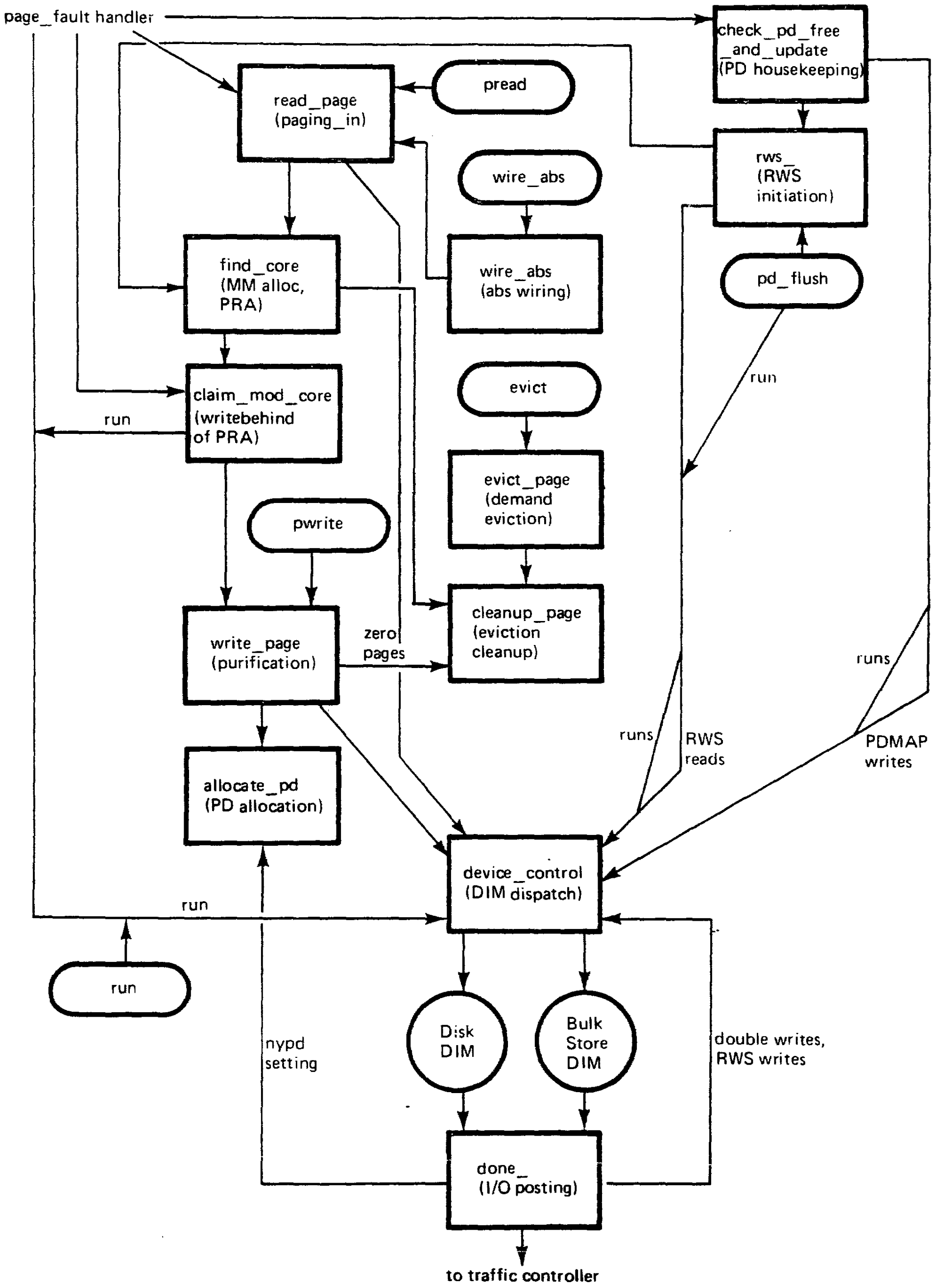

page_fault handler
check_pd_free_and_update (PD housekeeping)
read_page (paging_in)
pread
wire_abs
rws_ (RWS initiation)
find_core (MM alloc, PRA)
wire_abs (abs wiring)
pd_flush
claim_mod_core (writebehind of PRA)
run
evict
evict_page (demand eviction)
run
pwrite
write_page (purification)
zero pages
cleanup_page (eviction cleanup)
runs
runs
RWS reads
PDMAP writes
allocate_pd (PD allocation)
device_control (DIM dispatch)
run
run
Disk DIM
Bulk Store DIM
nypd setting
double writes, RWS writes
done_ (I/O posting)
to traffic controller

Figure 8-8. ALM Page Control Call Flow

Main Memory Frame Allocation

(find_core in page_fault)

Perhaps the most fundamental interface of all is that which finds a free main memory frame entry into which a page is read, including that performed on behalf of a page fault. This is the routine find_core, in the program page_fault. This routine is invoked by ALM page control whenever a frame of main memory is needed, other than some specific frame (abs-wiring).

The basic mechanism of allocating a main memory frame is the running of the main memory replacement algorithm, which runs exactly as described in Section V. If there are free frames available, the program that freed them moved them to the head of the "used list," and the pointer sst.usedp points to a free frame. If none are free, the used list is searched for a frame that contains a page that can be evicted without any I/O, that has not been recently used. Frames that do not meet these criteria are moved to behind the pointer. Wired and abs-wired frames are skipped too. If fifteen frames are passed over because of the fact that they would need I/O to evict their pages, claim_mod_core, the purifier of pages, is invoked, which starts those I/Os, and the scan continues. It may be so that claim_mod_core found some pages of zeros, or caused the DIMs to call the interrupt side, in either case putting claimable pages ahead of the used-list pointer. If a tremendous number of frames are rejected, the system is crashed with the message "out of core" (main memory).

When a frame is found which meets the criteria, an attempt is made to evict it. This attempt consists of turning off the bit ptw.df, which allows the hardware to use the page, clearing the associative memories of the system, and testing to see if the page was modified any time in the interval between the original decision that it was not modified (hence no I/O was necessary to evict it) and this clear of the associative memories. If it indeed was modified in this window, the eviction has failed, the access bit (ptw.df) is restored, and the search for an acceptable page continues. If it was not, the eviction is successful, and cleanup_page (see "Eviction Cleanup," below) is invoked to complete the eviction. The core map entry is left behind the used pointer (most recently used frame), with the field cme.ptwp being zero, this indicating the fact that it is free. The core map entry representing the frame made available is designated by the value of index register 4, on the return from find_core. Since find_core inspects many PTWs, and may call claim_mod_core, which may involve many PDMEs and CMEs and PTWs, no index registers are preserved by find_core.

The reader should note that pages that require updating to the paging device, even though they are not modified, require I/O to be evicted, and are thus not acceptable to find_core.

Replacement Algorithm Writebehind

(claim_mod_core in page_fault)

The main memory frame allocation function avoids frames containing pages that require I/O for their eviction so that it can return a usable page frame to its caller in minimal real time, allowing the read operation that the caller is sure to initiate to be started as soon as possible. This allows all writing on behalf of the replacement algorithm to be initiated while the read is in progress. This starting of writes is performed by the subroutine claim_mod_core in page_fault. This subroutine is invoked at the end of every page fault. When the main memory frame allocation function is invoked on behalf of some other action than a page fault, it is not invoked. In this case, the next page fault simply causes claim_mod_core to consider a larger set of pages than otherwise. The subroutine claim_mod_core is also invoked by find_core when fifteen frames have been skipped because of the need to perform I/O to accomplish their evictions.

Three functions are performed by claim_mod_core; any page frame skipped by find_core because of the need to do I/O to effect its eviction (whether actually modified or simply not yet on the paging device (nypd)) has such I/O started upon it. This is done via a call to write_page, the page writing/purifying function. Pages with their used bits on have them turned off; this is normally a function of the replacement algorithm, but the latter (find_core) must leave these bits on so that claim_mod_core will not initiate writes on pages that ought not to be evicted in the near future. A check is made to determine that no more than thirty writes are outstanding (page control disk writes only, not VTOCE writes), and the DIMs are "run" (see "DIM Interface and 'Running'" earlier) until this is so. This third function ensures that find_core is not processing vast numbers of frames because a very large number of writes have not completed.

The routine claim_mod_core processes all frames from the point it last left off (indicated by sst.wusedp) to the tail of the used list (where find_core is now, indicated by sst.usedp). Since calls to claim_mod_core might call the page-write function to start writes, and this might involve calling DIMs, which might call the interrupt side, the state of the pointer sst.usedp and the position of individual frames in the list maybe affected by invoking claim_mod_core. Also, claim_mod_core preserves no index registers.

Page Writing/Purification

(write_page in page_fault)

The contract of the page writing/purification function (the routine write_page in page_fault) is to start only I/O necessary to ensure that there is a good copy of a page outside of main memory (excepting the case where the page becomes modified after invocation of this routine). If some function, such as the deactivation-time service (pc$cleanup) wishes to purify the main memory page unconditionally, it must take steps that no process can reference the page (i.e., setfaulting all of the SDWs, as the deactivation function of segment control does). Purification consists of making the copy in main memory "pure," i.e., not modified with respect to secondary storage or paging device (whichever is appropriate).

The basic task of write-page is to initiate an I/O operation, writing the page out. The peripheral tasks consist of making all of the state transitions upon the PTW and CME of the page and page frame to set them 'out-of-service' (meaning "I/O in progress," <u>not</u> unusable), checking for all zeros, and checking whether allocation of a paging device record is in order.

The routine write_page is also used to cause the writing of unmodified pages (pure pages) which are not on the paging device (ptw.nypd) to the paging device. Thus, write_page performs precisely that function required by the replacement algorithm to make a page evictable without any I/O. It can also be seen that write_page is invoked on pages that are both modified and unmodified in main memory. The stack variable "mod_flag" tells what case is true (zero = not mod).

When invoked on a modified page (ptw.phm is on), write_page checks for a page of all zeros (unless the switch aste.gtpd is on to inhibit this). Such pages are evicted at this time, by write_page, calling cleanup_page to finish the eviction. As in find_core (see "Main Frame Allocation" above), a two-step trial eviction is necessary. When it has been determined that a page is all zeros, access is removed (ptw.df set off), the associative memories of the system cleared, and the page checked for zeros. If found not to be zero at this second check, it is treated as though it were not found as zero the first time. Any PD record associated with such pages are freed by a call to pd_util$pd_delete_. At this time, the disk record address of the page is <u>nulled</u> (see Section VII, "Address Management"), holding it for either later resurrection or deposition by segment control.

The routine write_page turns off the modified bit in the PTW, (ptw.phm) using a lock-type instruction (ANSA). The PTW associative memories of all processors are then cleared. If the page is modified before the clear of associative memories, but after write_page noted that the page was modified, the data bases will be modified to indicate that the page was modified (such as pdme.mod), and the write will proceed in any case. If any processor modifies the page <u>after</u> the clear of associative memories, the next attempt to evict the page will find that it was modified, and thus the copy written by this invocation of write_page was invalid. Note that access to a page remains <u>on</u> during a write.

If invoked on a modified page, the routine write_page turns on "file modified" switches (aste.fms) in the ASTE of this segment and all superior directories, unless aste.gtms is on, inhibiting this. (See the description of this flag in Section II.) The routine write_page also rethreads the PDME for a page which it writes (via a call to pd_util$rethread) to the tail (most recently used) position of the PD used list. This is to implement the part of the paging device management algorithm (see Section V) that states that pages in main memory are to be considered among the most recently used.

One very critical action of write_page is to check if the page being written must be allocated a paging device record; this check is made by the subroutine allocate_pd (described below) in all cases (other than a zero page). Whether or not the page is modified, allocate_pd allocates a PD record if it should, if it can, and one is not already allocated for this page.

The final action of write_page is to invoke device_control$write to actually call the appropriate DIM to start a page write. Since it is part of the DIM interface (see "DIM Interface" earlier in this section) that the request being issued may even be completed during that call, this must be the last action taken by write_page.

The routine write_page is invoked with index register 4 pointing to the CME of the frame that is to be written. It expects index 2 and pointer register 2 to describe the PTW of this page, and index 3 the ASTE of its segment. All of these registers will be preserved. No statement is made about the final state of the page or frame, or whether or not it will be out of service upon return from write_page.

The routine write_page is normally invoked from ALM page control, on behalf of claim_mod_core, with the registers set as above. However, it may also be invoked as page$pwrite from call-side PL/I code. In this case, the interface routine page_fault$pwrite is invoked, which establishes the ALM page control environment, and the necessary pointers and index registers, and calls the routine pwrite.

Page Reading

(paging-in) function - (read_page and read_page_abs in page_fault)

The basic task of the reading function is to bring a page of a segment into main memory; if null or nulled, a page of zeros will be created. Generally, a read of disk or paging device will be required, and the page-reading function will initiate this read. The page reading function indicates to its caller whether or not waiting will be required by this caller.

It is part of the task of the page-reading function to check that both adequate record quota and adequate disk storage space are available to accommodate the page. Quota must be checked each time a page with a null or nulled address must be paged in for it is at that time quota is charged. Physical device allocation must be checked each time a page with a null address is paged in. If either of these operations cannot be successfully performed, i.e., adequate allocations do not exist, action can be taken before the page is created in main memory. It is only legal for the fault side to encounter out-of-quota or out-of-physical volume situations; all segments treated by the call side should be quota-inhibited and prewithdrawn.

The page-reading function has two entries, read_page and read_page_abs, in the module "page_fault." The usual entry is read_page, which, as part of its task, locates a main memory frame into which to read the requested page, via a call to the main memory frame allocation function find_core. The other entry, used only by the abs_wiring function, is supplied the identity of a specific main memory frame into which the paging-in is to be done. Either entry expects to be called, via TSX7 instruction, with index register 2 set to the relative address of the PTW of the page to be read in, and index 3 to the relative address of its AST entry in the SST. The routines return with not only these registers set, but index 4 set to the relative address of the core map entry of the main memory frame into which the paging-in was done. The routines return to the location past the TSX7 instruction if they started I/O that has not been completed, in which case the upper A register has the event ID to wait for. Otherwise, if no incomplete I/O exists, or none was started at all, a return to the location two locations beyond the TSX7 is executed.

If the page-reading function encounters a page on which a read-write sequence (RWS) is in progress, the caller is returned the event ID of that RWS. This will cause the call side to ultimately set a notify-requested bit in the affected PDME. The fault side will initiate an RWS abort (turning on pdme.abort) in this case.

Other actions of the page-reading function include maintaining the current-length and records-used ASTE parameters of the segment in the case where a page is created (zero page paged in), and performing the CME and PTW state transitions associated with setting a page out of service in other cases. When the page being paged in is on the paging device, the associated PDME is rethreaded to the tail of the PD used list, in keeping with the policy that all pages in main memory are among the most recently used on the paging device. As is the case with the page-writing function, the actual call to device_control (in this case device_control$read), the DIM dispatcher, must be the last action taken, for the page may not even be out of service on return from this call.

A major consideration of the page-reading function is to loop, redefining its arguments, when the call to the disk-record allocator (free_store$withdraw) indicates that a page of the FSDCT must be paged in to perform the allocation. As explained under "FSDCT Paging" earlier, the page reading function must redirect itself to page in a page of the FSDCT instead of the page passed as an argument, when paging in that page involves allocating a disk record, and that allocation requires paging in the FSDCT. In this case, whatever wait event or lack thereof results from such activity will be returned to the caller of read_page (or read_page_abs) to wait on.

The page-reading function is normally invoked at the read_page entry point, via the routine "readin" in the page-fault handler (see "Page Fault Handling" in Section IX). However, it may also be invoked from page$pread from call-side PL/I code, such as the process-loading function. In this case, the interface routine page_fault$pread is invoked, which establishes the ALM page control environment, and calls read_page. This interface routine conveys the wait-event ID returned by read_page to its caller, returning zero if there was none. However, in the case of bulk store I/O, the interface routine page_fault$pread "runs" the bulk store DIM in a loop to await the completion of the I/O. This is to obviate the need for a separate bulk store waiting mechanism for the process-loading function. Thus, only disk I/O or RWS events are returned to the caller of page$pread.

Paging Device Record Allocator

(allocate_pd in page_fault)

The paging device record allocator is invoked at two times; at the completion of a disk read, and during the page-writing function. Its task is to determine whether a page is or should be on the paging device. If the latter is the case, either the bit ptw.nypd is set (if invoked on behalf of a disk-read completion) or the page is actually migrated to the paging device (if invoked from the page-writing function).

Migrating a page to the paging device consists of finding a free paging device record, and updating the CME and PTW associated with the page being migrated, as well as the PDME for the free record found, to indicate that they are all associated with the same page. The routine allocate_pd performs an alternate return depending on whether or not it migrated the page to the paging device as a result of this invocation.

The decision as to whether a page should go on the paging device involves the decisions as to whether it is already there, whether the segment to which it belongs has the "global transparent to paging device (gtpd)" attribute, explicitly inhibiting this action, whether or not there is actually an enabled paging device in use, and the consideration of the ptw.first usage-optimizing feature (see the description of this bit in the PTW breakdown in Section VI.

When invoked on behalf of the completion of a disk interrupt, the page is not actually migrated to the paging device unless the "pd_writeahead" switch is set in the SST (sst.pd_writehead; see the description of this field in the SST breakdown in Section VI. This feature is not currently operative.) Rather, the bit ptw.nypd in the PTW is turned on. This bit tells the main-memory replacement algorithm that the page-writing function must be invoked to evict this page, allowing it to skip that page in a search for the "most available" page to evict. When the page-writing function is called for this page, on account of this bit, it will cause the paging device record allocator to be invoked once more at which time the page will actually be migrated to the paging device.

When the paging device record allocator actually decides to migrate a page to the paging device, there should be free records available on the paging device. The paging device management algorithm attempts to keep a free pool by ensuring the existence of a small fixed number free or being freed at the beginning of the processing of each page fault. Thus, the free paging device record at the head of the paging device used list is normally allocated to the page on behalf of which the paging device record allocator is being invoked. If the record at the head of the paging device used list is not free, an action known as a "PD Desperation" is performed. This action, performed by the PD Desperator, pd_util$force_get_pd, consists of walking down the PD used list no more than fifteen steps to find a paging device record whose page is evictable without a read-write sequence, or an eviction from main memory. A read-write sequence (RWS) may not be performed at this time; the call history of the paging device record allocator may well include the main memory frame allocation function, which is necessary to initiate an RWS, and is not recursive. Furthermore, the completion of the RWS could not be awaited at this time; ALM page control does not wait, but indicates wait events to its caller.

If a PD desperation fails, the paging device record allocator fails (see the SST breakdown in Section VI for the names and meanings of meters of this event), and the page is not migrated to the paging device. This causes the page-writing function to turn off any "nypd" PTW bit which may be on, causing all attempts to migrate the page to the paging device for this activation to be abandoned.

The paging device record allocator expects to be called via a TSX7 with pointer register 2 and index register 2 describing the PTW of a page for which paging device allocation must be checked and/or performed. Index register 3 must point to the AST entry of the segment containing the page. The page must be in main memory, and not out of service or undergoing a read-write sequence (RWS), and index register 4 must describe the core map entry (CME) for the main memory frame it occupies. The stack variables "devadd" and "did" must contain the record address (in the format described at the beginning of Section VI) and the PVT index for the page.

The paging device record allocator returns to the location beyond the TSX7 if it did not allocate the page to the paging device as a result of this invocation, and two locations beyond if it did. If it migrated the page to the paging device, the stack variables "devadd" and "did" will be modified to reflect the paging device record address of the page, as well as the core map entry of the page.

RWS Initiator

(rws_ in pd_util)

The RWS initiator is supplied the identification of a paging device record (as a relative pointer to its PDME) and is responsible for starting a read-write sequence (RWS) on that PD record. It invokes the main-memory frame allocator (find_core in page-fault) to allocate a frame for the RWS, and the DIM dispatcher device_control$read to start the read cycle of the RWS. It threads the CME of the main memory frame and the PDME of the paging device record out of the main memory and paging device used lists, respectively, and performs the necessary state transitions upon all of these objects to indicate that the read cycle of an RWS is under way. The RWS initiator neither awaits completion of the read cycle nor initiates the write cycle; the former is done by either the PD replacement function or the interface routine pd_util$pd_flush, the latter is done by the interrupt side.

The RWS initiator never allows more than thirty RWSs to be outstanding; when it has initiated the thirty-first RWS, it "runs" the DIMs until one of them has competed (i.e., the count sst.pd_wtct has gone down).

As with the page-reading and page-writing functions, the call to device_control to actually start the reading I/O is the last action performed by the RWS initiator, as the RWS it initiates could be over (especially on account of error) by time the return from this call is complete.

The RWS initiator is called via a TSX7 instruction from ALM page control. It expects index register 1 to point to the PDME for the PD record to undergo RWS. It saves no registers, it has no alternate returns. It destroys the contents of the stack variable "ptp_astep," used by the page-reading function, among others.

The RWS initiator is used by the PD replacement function and a large number of call-side functions, such as the deactivation-time service and the PD reconfiguration function. In these latter cases it is called as page$pd_flush from PL/I code, which invokes the interface routine pd_util$pd_flush. This interface routine establishes the ALM page control environment, and invokes the RWS initiator upon the PDME located by the PL/I pointer argument to this routine. The interface routine also awaits the completion of the read cycle of the RWS initiated, by "running" the bulk store DIM. This maintains the convention that no RWS read cycles may be in progress at the time the page table lock is unlocked.

Paging Device Housekeeping and Replacement

(check_pd_free_and_update in pd_util)

This function serves to keep a small pool of free paging device records available for the paging device record allocator at all times. The paging device record allocator cannot free records on demand except in certain special cases. The paging device housekeeping function also serves to write out the paging device map to the first few records of the bulk store every second. This copy is maintained for the use of the post-crash PD flush (see description of that in Section IX).

The paging device housekeeping function is invoked at the beginning of the processing of every page fault, from the page-fault handler. It initiates the writing of the paging device map if that has not been done within the last second; this map writing is done with the "no_interrupt" flag to the DIM set on. The interrupt side of page control does not want to be informed when this I/O has completed.

The PD housekeeping function implements the paging device replacement algorithm outlined in Section V. The paging device used list is scanned from least-recently-seen-used to most-recently-seen-used end until ten records are free or in the process of undergoing RWS. An RWS is initiated for each PD record passed which is modified with respect to disk; the RWS initiator just described is used. Each record inspected that is not modified with respect to disk is freed; its page contents are migrated off the paging device by modifying the PTW of that page. This PTW currently describes this PD record. It is made to describe the disk record currently described in the PDME. Pages that are found, by inspection of the PTW designated by the PDME field pdme.ptwp, to be in main memory, cause their PDMEs not to be claimed, but rather, made to be "recently seen as used" by rethreading them to the tail of the PD used list. This is in keeping with the policy that those PD records seen in main memory are to be considered among the most recently used.

The final action of PD housekeeping is to check that no RWS read cycles are in progress. RWS read cycles may not be in progress when the page table lock is unlocked, and it is the responsibility of whichever agency invokes the RWS initiator to see that they complete before it exists. The strategy of waiting for the (bulk store) reads to complete all at once allows the RWS initiator to start all of these reads in parallel. This overlap optimizes performance via the queueing facility of the bulk store DIM. The PD housekeeping function "runs" the DIMs in a loop until no more RWS read cycles (counted by sst.rws_reads_os) are outstanding. During this looping, the bulk store DIM will invoke the interrupt side to initiate the RWS write cycles.

The PD housekeeping function is invoked via a TSX7 instruction from ALM page control. It preserves no registers, and has no alternate returns.

Eviction Cleanup

(cleanup_page in page fault, and code in pc$cleanup and pc$truncate)

The eviction cleanup function consists of modifying all page control data bases necessary to indicate that a page has been evicted from main memory. This function does not include the actual turning-off of the PTW access bit, ptw.df. The latter involves associative-memory and cache clearing, and turning it back on if the page was found to be modified after the clear had taken effect. It is the responsibility of the eviction cleanup function to modify all other data objects once access to a page has successfully been turned off. In ALM page control, this function is performed by the subroutine cleanup_page in page_fault. This routine is invoked by the main-memory replacement algorithm, when it evicts a page, and by the demand-eviction and abs.wiring functions (see description later in this section) when evictions are performed.

The call side also evicts pages, in the routine pc$cleanup invoked on behalf of the segment control deactivation function, and in the truncation function. PL/I code in these routines performs work similar to that of cleanup_page. This work is much simpler in the case of truncation.

Eviction cleanup consists of maintaining the AST entry of the segment and freeing the main memory page from which the page was evicted. The number of pages in main memory is updated. If the page evicted contained zeros (i.e., its address is now nulled, the "number of records used" of the segment must be adjusted, as well as the record quota account against which the segment's pages are charged. If no more pages of the segment are in main memory after this eviction, the ASTE "init" bit (used by the AST replacement algorithm) must be turned on. If the highest-addressed page of the segment which is nonnulled/null or was in main memory was evicted, the current length of the segment is adjusted. The disk or PD address from the core map entry of the frame from which the eviction is being performed is placed back in the PTW for the page. The PTW "first" bit (for the optimizing algorithm described under the description of sst.ptw_first in Section VI is turned off, indicating that the page has been evicted at least once from main memory since activation.

The routine cleanup_page is invoked via a TSX7 instruction from ALM page control. It expects pointer register 2 and index register 2 to describe the PTW of the page being evicted, and index 4 to describe the CME of the main memory frame from which it is being evicted. It will preserve these registers, as well as set index 3 to the ASTE. There are no alternate returns.

Per-Page Cache Management

(cam_cache in page_fault)

The general strategies for managing the Multics Processor caches are described under "Encacheability Control" in Section II. These strategies cover modification of main memory by several processors, and by all I/O devices except I/O devices used for paging. The per-page cache management strategy covers these latter cases; when a page is read in from paging device or disk, the contents of main memory locations which may be in processor caches will be modified without changing the contents of these cache locations. The avoidance of this situation is the goal of the per-page cache management strategy.

Paging I/O has the unique property that a main memory frame into which a page of a segment is being read is guaranteed to contain no information that any processor (or process) can access. Therefore, if it could be ensured that no words of that page appeared in any processor's cache at the time the read was begun, there would be no chance that the reading in of data by the IOM could contradict any data in a processor cache. Thus, it would be adequate to clear the caches of all processors at the time a page of a segment was evicted from main memory, i.e., made inaccessible to the processors of the system.

The Multics processor cache includes a feature known as "selective clear," a hardware mechanism for iterating through all the columns and blocks of the cache, and invalidating the contents of any block that contains information from a given page. This mechanism is available via the CAMP instruction, with the "4,du" bit on in its effective (internal) address. The frame is identified by the upper bits of this address. This instruction also clears all processor PTW associative memories, which is desired at page eviction time. Thus, at page eviction time, all system processors are forced to execute an instance of this instruction to clear all words of the page being evicted out of their caches, and all PTWs out of their PTW associative memories. The instance of this instruction so constructed is stored in scs$cam_pair; the general CAM/connect strategy is described in the Multics Reconfiguration PLM, Order No. AN71.

The function available to ALM page control as the "cam_cache" routine is also available to PL/I code as page$cam_cache, which invokes the interface routine cam_cache_ext in page_fault. However, most PL/I code calls page$cam before unlocking the page table lock, which clears all system caches and associative memories totally.

It is critical to this strategy that the abs_segs used by page control to check page frames for zeros not be encacheable.

Demand Eviction

(evict_page$evict_page)

The demand page eviction function is one called by the main-memory deconfiguration function (see Section IX), on behalf of the system reconfiguration software, and on behalf of the I/O buffer abs-wiring function. In the latter case, it is used to evict the previous resident of a main-memory frame into which a page of an I/O buffer segment is going to be abs-wired. It is also the responsibility of the demand eviction function to inform its caller of any RWS or page transfer I/O in progress in the main memory frame being vacated (having a page evicted from it).

The demand page eviction function is called as page$evict_page from PL/I code only. It is called with a PL/I pointer to the core map entry representing the main memory frame from which it is to be vacated. It returns a wait event ID; if that is zero, it has successfully vacated the frame, if not, the caller must await that event (via the multiplex wait protocol and the call-side wait-coordinator) and call evict_page again when the event has happened. The demand page eviction function is an excellent example of those functions that perform successive state transitions upon page control objects, which must be constrained from retrogression via the setting of control bits. In this case, the caller of the demand-eviction function must have set either of the CME bits cme.removing or cme.abs_w, to ensure the success of the vacating (prevent the main memory frame allocator from allocating the frame).

The demand page eviction function begins by checking that no RWS or ordinary page-transfer (ptw.os on) I/O is in progress in the frame being vacated; if so, the caller is returned the event ID corresponding to the operation in progress. If, via the multiplex wait protocol, the caller chooses to wait for this event via the call-side wait coordinator page$pwait, the latter will turn on the appropriate notify-requested bits to cause the interrupt side to notify the completion of these events. If there is no I/O going on in that frame, the demand page eviction function will successfully complete in this call, i.e., there will be no waiting.

If the page in the main memory frame is wired (but may not be abs_wired), it must be moved to another main memory frame, in such a way that it is never made inaccessible to the system processors. Since the page may be being modified by the system processors there is no way to move the page while other processors are accessing it. Furthermore, it is not desirable to change the contents of the PTW, which will be necessary, while other processors are using it. Thus, a mechanism is provided to halt all of the system processors except the one executing this code, until this processor releases them. This service is provided by the CAM/connect mechanism, which sets appropriate flags in the SCS segment when this is the case. First, the main memory frame allocator (find_core) is invoked to obtain a page frame into which to move the wired page. The state of the "modified" bit (ptw.phm) of the page being moved is saved, and it is turned off. This must be done in one unitary (key-line) operation, lest a modification between the inspection and the turning-off be lost. All of the system processors, except the one executing, are then stopped, and all PTW associative memories cleared, via a call to cam_with_wait in page_fault. This routine also causes all words of the old frame to be selectively cleared out of all the caches of the system processors. The contents of the old frame are then moved to the new frame via the use of two non-encacheable abs-segs. If, after so doing, the PTW "modified" bit (ptw.phm) has not come on (since it was turned off) the contents of the old frame and new frame are the same. If not, the contents are moved once again (this is metered by sst.recopies). The contents cannot now possibly change, since all processors are halted. The possible modification just noted is then "or'ed" into the PTW "modified" bit (ptw.phm), and the PTW main memory address (ptw.add) is changed to describe the new frame. The system processors are then released via zeroing the cell scs$cam_wait, on which they all are looping. The CME for the old frame is made to be free (although it has one of the bits cme.abs_w or cme.removing protecting against its accidental claiming), and the CME for the new frame is made to describe the page moved, which had been described by the CME for the old frame.

If the page in the frame being vacated is not wired, then the task is vastly simplified, as it is permissible to make the page inaccessible. This is precisely what is done. The PTW access bit, ptw.df, is turned off, and the system PTW associative memories are cleared, and the caches selectively cleared, as for any eviction. If the modified bit is not on after this clear, (it could not have been on before access was removed, or it would still be on), then a successful eviction has just been performed. The eviction cleanup function (page_fault$cleanup_page) is invoked to complete the details of the eviction, and the frame has been successfully vacated. If, on the other hand, the page was modified, either before access was turned off or after, we must move its contents to another frame, which is cheaper and faster than starting an I/O and causing the caller to wait for it. It is also deterministic; there is no telling how many times the page could be modified while the caller waited for it, while moving the page avoids the entire issue. Thus, the main memory frame allocation function (find_core) is invoked to obtain a frame, and the page is moved. System processors do not have to be halted, as opposed to the wired case, as the page was just made inaccessible, and the associative-memory clear mechanism ensures that no processors are left accessing it. The contents of the PTW address field (ptw.add) is changed to describe the new frame. The CME for the old frame is made free (although it is protected by cme.removing or cme.abs_w) and the CME for the new frame is made to describe the page moved. The access in the PTW is restored, i.e., ptw.phm is turned back on, and processors continue to use the page in its new location (although, however, if they attempt to access it before this, but after the time that access was revoked, such processors caused their processes to take a page fault, and wait for the page table lock, now held by this process). Again, the eviction is complete with no waiting.

Page abs-wiring

(evict_page$wire_abs)

The page abs-wiring function is used only by the segment abs-wiring service (in pc_contig, described in Section IX. It is invoked from PL/I page control only, given a page (as an ASTE pointer and page number), and a free CME (usually vacated via the demand-eviction function just described) into which to abs-wire the page. It assumes that the caller has set the bit cme.abs_w in the CME for the frame participating in the abs-wiring, to prevent any page other than the one being processed from coming into that frame. If the "wired" bit of the PTW for that page is not on, the abs-wiring function turns it on, indicating that the page, wherever it might be now, or wherever it may come, may not be evicted.

The abs-wiring function is among that class of ALM page control functions that either complete their task when called, or return a wait event on which the caller must wait, and call that primitive back when the event has happened. The abs-wiring function thus returns a zero wait event ID if it has unsuccessfully placed the page in the frame into which it is to be abs-wired, or an event ID on which to wait.

The first check made by the abs-wiring function is that the page is not already involved in I/O (ptw.os on). If so, the caller is made to wait for the completion of that I/O. Since ptw.wired has been turned on, a page out of service on a read will not be evicted once it comes in, and a page out of service on a write will be selected for no further writes by the main memory replacement algorithm.

If there is no I/O on the page in progress, two different cases occur for the cases of the page being in main memory already (may even be in the required frame already, via previous calls), or not in main memory. If it is in main memory, and in the required frame, the task is complete, and a zero event ID is returned to the caller. If not, the page must be moved from its current frame to the new frame. The task is identically that of the code in the demand page-eviction function which moves wired pages, as this page is now wired. This code is used, the page is moved (including halting all processors, etc.), and the abs-wiring is complete. If the page is not in main memory, the page-reading function (read-page-abs) is invoked to read the page in. As described under the description of page reading function (earlier) the caller of that function (and thus, in this case, the caller of the abs-wiring function) is made to wait for the completion of FSDCT paging I/O, RWS completion, or page-reading, whatever may be the case, and retrying through repeated calls. Thus, this path through the abs-wiring function uses the multiplex wait protocol of the caller to drive the FSDCT paging mechanism and await RWS completion transparently to the mechanism of the abs-wiring function.

Note that the version of the page-reading function used by the abs-wiring function (read_page_abs) does not allocate a main memory frame, but uses the specific main memory frame supplied by the caller, in this case, evict_page$wire_abs.

I/O Posting

(the Interrupt Side) (page_fault$done_)

The function of posting (processing the completion) of paging I/O operations is one of the most critical in page control. It includes performing all of the state transitions out of I/O or RWS states of page control data objects, all error processing, and the initiation of RWS write cycles and double-writes, and calling the traffic controller notify primitive.

The I/O posting function is implemented in the routine done in the module page-fault. It is invoked solely from the storage system DIMs when they notice I/O completions. It is accessed directly via TSX7 from the bulk store DIM, and via the interface page$done, which leads to the interface routine page_fault$done, by the disk DIM. The routine done_ is invoked in the ALM page control environment with the page table lock locked. Its parameters are the stack variables "core_add" and "errcode," containing the main memory address and status of the I/O operation which was completed.

One critical function of the interrupt side is to resurrect disk addresses upon the successful completion of RWSs, double writes, or other disk writes. As described in Section VII, this resurrection signifies just this successful disk writing, indicating that the address may safely be reported by pc$get_file_map to a VTOCE.

The interrupt side begins its task by looking at the core map entry indicated by the main memory address passed to it. It must either indicate an RWS in progress, or designate a PTW which has its out-of-service (ptw.os) bit on. It may designate a free main memory frame. We consider first the case of normal paging (non-RWS) I/O completion.

If a page read completes, a check is made to see if it completed unsuccessfully (with an error). The error actions, here as elsewhere, are described in "Error Strategy" earlier in this section. Assuming there was no error, the CME for the page frame of main memory is threaded back into the used list, as "most recently used." The out-of-service bit is turned off. The paging device record allocator is invoked to cause the "not-yet-on-paging-device" bit to be set if necessary. If the notify-requested flag was on in the CME, the special traffic controller interface pxss$page_notify (see "Traffic Controller Interface" earlier in this section) is invoked to notify the completion of the read.

In the case of a page write completion, with no error, the CME is threaded back into the used list as "least recently used" (best candidate for eviction), and the out-of-service bit in the PTW (ptw.os) turned off. If the write was for a page not on the paging device, the disk address is resurrected (made live, not nulled), and the bit aste.fmchanged turned on to trigger a VTOCE update. If it was a write for a page on the paging device, the PDME is inspected (pdme.double_writing) to see if it was a double-write (write to disk for a page on the paging device). Otherwise, it was a write to the paging device. If it was a write to the paging device, a check is made to see if a double-write should be initiated, based upon the properties of the page, the segment, and the double-writing control switch (sst.double_write) in the SST. (See the description of that switch in Section VI for the various decisions and interpretations.) if a double- write is to be started, the page is put back out-of-service, the threading-in of the core map entry avoided, and a call made to the DIM dispatcher device_control$write to start the I/O. The bit pdme.double_writing is put on before this call is made to indicate to the interrupt side what action should be taken at the completion of that I/O. If double-writing is not to be started, the CME is threaded as stated, and the traffic controller called to notify the write-completion if required. If a double_write's completion was noted, the double_writing PDME flag (pdme.double_writing) is turned off, the disk address resurrected, and the PDME marked as not modified with respect to disk. Again, an optional notify follows. These actions, as the rest of the interrupt side posting logic (other than error handling) are shown in Figures 8-9 and 8-10.

The actions taken for the completion of RWS I/O depend on whether it is a read or write cycle that has completed. When a read cycle has completed (cme.io tells which), the write cycle is started by setting the bit cme.io, and starting the I/O for the write cycle. The CME and PDME involved remain out of their respective lists, and no notifications are performed.

The completion of the write cycle is more complex, as it implies the completion of the entire RWS. At the start, notification of the RWS event is performed via the special traffic controller entry pxss$rws_notify if any of the bits pdme.notify_requested, pdme.removing, or pdme.abort are set; any of these bits implies that some process is waiting for the RWS event. Assuming no error, (see "Error Strategy" for discussion of the error path here), the disk address in the PDME is resurrected, indicating successful transfer of the data to disk. If no abort (page fault by a process while the RWS was in progress) was observed (pdme.abort would have been set on by the fault side), the PTW for the page which underwent RWS is located from the PDME, and changed to contain the disk address from the PDME (it now contains a paging device address). The PDME is zeroed, and marked as free, being put into the PD used list. The main memory frame is similarly freed, being put into the main memory used list. If however, a post-crash PD flush was responsible for initiating the RWS (pdme.flushing on), the PTW (none exists) is not adjusted, nor is the PDME cleared or freed. Rather, the PDME flushing, RWS, and modified flags are turned off. This leaves the PDME intact for the call side to inspect, so that an error during the RWS can be determined to have happened or not by inspecting the nulled/live status of the disk address (pdme.devadd) in the PDME.

If an RWS abort was noticed, the main memory frame in which the RWS occurred is converted into a normal page-holding frame. The ASTE of the relevant segment is adjusted to indicate the proper number of pages in main memory, etc.; and the CME pointers are set to describe the PTW and ASTE. The RWS flags in the CME and the PDME are turned off. The "modified" status of the PDME, which has never been turned off, remains in effect. The PDME is put back in the PD used list. The CME is put back in the main memory used list, in most-recently-used position. The process which had turned on the abort bit, causing the abort, has already been notified, and is now either "ready" or waiting for the page table lock.

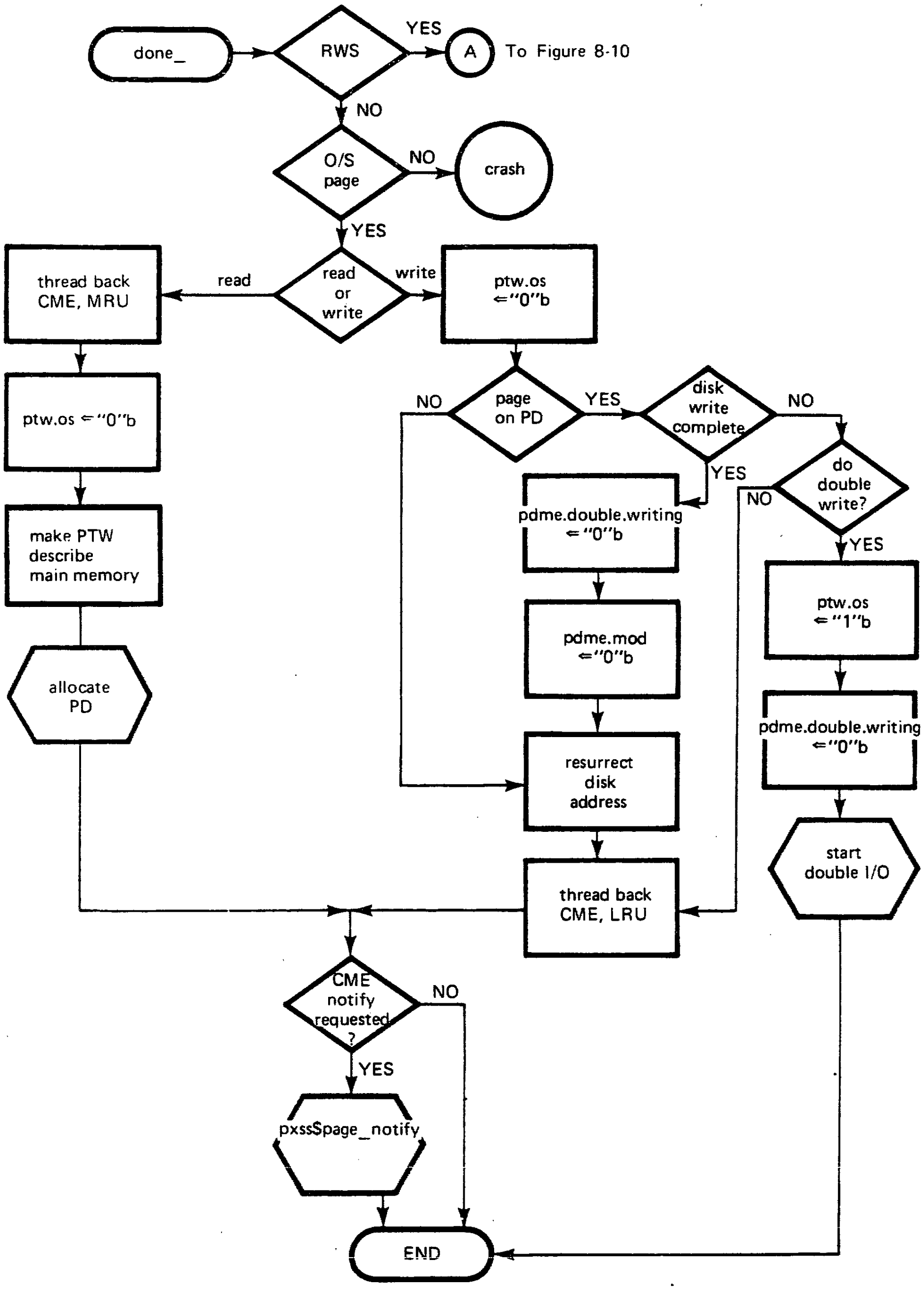


Figure 8-9. Page Control Interrupt Side, normal posting

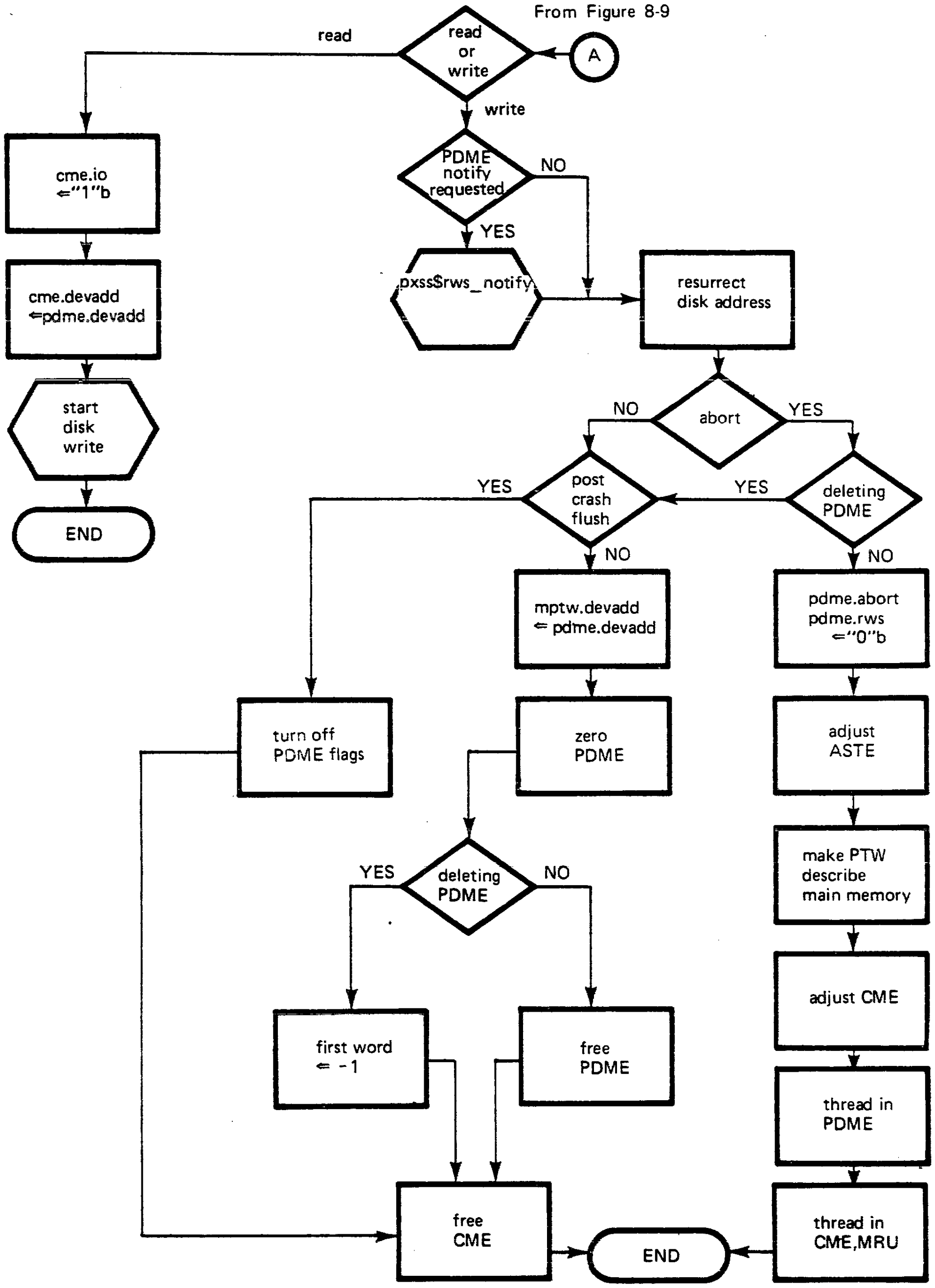


Figure 8-10. Page control Interrupt Side, RWS posting

Utility Subroutines

This discussion provides brief descriptions of the utility subroutines in ALM page control. All of these subroutines are in the modules page_fault and pd_util. A utility subroutine, by this definition, is a routine that does not affect the state of page control objects; PTWs, CMEs, PDMEs, other than perhaps rethreading them. Any routine that performs state transitions is among the critical agents described under "Internal Interfaces" earlier in this section. The name, function, and calling sequence of each of these routines, with whatever comments are appropriate, is given.

init_savex

Called by TSX6, sets up page control index-seven save stack. Used to establish the ALM page control environment.

savex

Called by TSX6, saves index 7 in the index-seven save stack. Any routine that saves index 7 via this routine transfers to "unsavex" to return.

unsavex

Invoked via TRA, returns from a routine which called "savex" at its start. Pops the index-seven save stack.

thread_to_lru

Invoked via TSX7, with index 4 pointing at a core map entry. Given a CME in the used list, rethreads it to the head (least recently used) position, adjusting whatever global page control pointers are necessary.

thread_out

Invoked via TSX7, with index 4 pointing at a core map entry in the used list. Threads it out of the used list, zeroing its thread word, and changing whatever global page control pointers are necessary.

thread_in

Invoked via TSX7, with index 4 pointing at a core map entry _not_ in the used list. Threads it into the used list, and the head (least-recently used), updating global page control pointers.

thread_in_mru

Invoked via TSX7, identical to thread_in, but the CME is placed at the _tail_ (most-recently-used) end of the main memory used list.

thread_lru_ext

Is a PL/I callable interface to thread_to_lru.

set_up_abs_seg

Called via TSX6, with the main memory address of a page frame of main memory in the stack variable "core_add." Places a non-encacheable SDW for that page frame in the SDW slot for the segment "abs_seg1," and makes pointer register 0 (ap) point to it. This is used for checking for zeros and zeroing main memory frames on behalf of the page-writing and page-reading functions, respectively.

clear_ core

called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Fills the frame with zeros, on behalf of the page-reading function.

check_for_zero

Called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Sets the "zero" indicator register to indicate whether or not that frame of main memory contains all zeros, on behalf of the page-writing/purification function (see "Page Writing Function" earlier).

cam_cache

Called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Clears the PTW associative memories of all processors, and selectively clears the words of that main memory page out of their caches. Destroys index registers 0 and 1. Available to PL/I code as page$cam_cache.

cam_ptws

Called via TSX7, clears the PTW associative memories of all processors. Destroys index registers 0 and 1.

cam

Called via TSX7, clears PTW and SDW associative memories and all caches of all processors. Destroys index registers 0 and 1. Available to PL/I code as page$cam.

cam_with_wait

Called via TSX7 with the main memory address of a page frame of main memory in the stack variable "core_add." Clears the PTW associative memories of all processors, and selectively clears their caches of all words of that main memory frame. Furthermore, all processors except the executing processor are made to halt until the variable scs$cam_wait is zeroed by the executing processor. Destroys index registers 0 and 1.

reset_mode_reg

Called via TSX7 by the page fault handler, restarts the history registers and re-enables the cache after a (page) fault.

get_pvtx

Called via TSX7, with index register 3 pointing to an ASTE. Extracts the PVT index for that segment out of its ASTE, placing it in the stack variable "did" and in the accumulator.

get_astep_given_pdmep

Called via TSX6, with index 1 pointing at a paging device map entry (PDME). Determines the AST entry offset of the segment to which the page in the PD record described by the PDME belongs. Places it in index register 3, conventional register for ASTEs. May not be used during post-crash PD flush.

check_quota

Called via TSX7, with index 2 and pointer register 2 describing a PTW, and index 3 its ASTE, by the page-reading function. Returns if the page that is to be read in is not null and not nulled, or there exists a record of quota to support its being read in. Otherwise, transfers to "errquit" in the page fault handler to signal record quota overflow.

reset_quota

Called via TSX7, with index 3 pointing at the ASTE of a segment for which a page is being destroyed. Used by eviction cleanup to decrement "records used" for the quota account of some segment. The entry quotaw$cu_for_pc performs the same function for PL/I code in the program pc.

bump_quota

Called via TSX7 with index 3 pointing at the ASTE of a segment against which another page of record quota will be charged. Called by the page-reading function once it has determined that sufficient quota and disk space exist to create the page.

type_terminal_quota

Called via TSX6 from check_quota, bump_quota, and reset_quota, determines whether directory or segment page quota is involved, and whether the segment involved has quota checking suppressed (aste.nqsw on).

update_csl

Called via TSX7 when a page is found to be zero by the page-writing function. Recomputes the current length of the segment in the ASTE (aste.csl) by scanning the page table backwards.

In pd_util

pd_delete_

Called via TSX7; with index 1 pointing at a PDME, in the used list. The PDME is cleared and threaded to the head of the PD used list. It is assumed that the caller has evicted whatever page was in that record.

get_devadd

Called via TSX7, with index 1 pointing at a PDME. Creates a standard-format (see beginning of Section VI) paging-device record address for the PD record described by that PDME, returning it in the accumulator and the stack variable "devadd."

get_pdmep

Called via TSX7, with pointer register 2 describing a PTW. If this PTW describes a page that has a paging device record associated with it, returns two locations after the TSX7, with the relative offset of the PDME for the associated PD record in index 1 (conventional for PDMEs) and the upper half of the stack variable "pdmep." If not, returns indirectly (TRA 0,7*) through the first location after the TSX7, with index 1 and the upper half of the variable "pdmep" containing a -1.

pd_rethread

Called via TSX7, with index 1 pointing to a PDME in the PD used list. Threads that PDME to the tail (most recently used) position of the used list. If the switch sst.count_pdmes is on (see its description in Section VI), an esoteric form of metering is performed, recording in a histogram its distance down the list before rethreading.

pd_insert

Called via TSX7, with index 1 pointing at a PDME not in the PD used list. Threads it into the PD used list, at the tail (most recently used) position.

SECTION IX

SERVICES OF PAGE CONTROL

This section describes the services that page control performs for the system. Foremost among these is the handling of page faults. Other services are performed for segment control, traffic control, and other supervisor subsystems. Although all of these services have been briefly described in Section V, the descriptions in this section explain the implementation of these services in terms of the mechanisms explained in Section VIII.

Other than the page fault handler, whose main path encompasses most of ALM page control, and the post purge function used by traffic control, all of these services are implemented in PL/I programs that operate on segments or portions of segments, calling the interfaces described in the previous section on each affected page, and multiplexing resultant wait events. The main memory and paging device reconfiguration services operate on portions of main memory or paging device instead of segments, again calling the ALM interfaces on each affected frame or record, and multiplexing the wait events.

All of the services of page control to segment control are implemented in the single PL/I program "pc", which, as noted in the previous section, has some code duplicating or subsuming functions of ALM page control were convenient.

PAGE FAULT HANDLING

The single most important function of page control is the handling and resolution of page faults. This code is implemented in the program page_fault, at the label "fault", transferred to by the fault vector directly, after the fault vector code has stored the SCU data for the page fault in pds$page_fault_data.

The essence of the page fault handler is to locate the page that must be paged in, and invoke the page-reading function to allocate a main memory frame and read it in. If successfully read in, the SCU data (machine conditions) is restarted; if I/O is started but not complete, the process must be made to wait for a completion of the I/O. If any of various error conditions prevail, the process must be caused to signal an appropriate condition, or restart the page fault to take a segment fault.

The most difficult task of the page fault handler is to locate the PTW for the page faulted on. Between the time that the processor actually takes the page fault and the page table lock is successfully locked to this process, it is possible that a "setfaults" operation (destruction of SDWs) might be performed on the segment containing the page, or the page of the descriptor segment containing the SDW for the segment might be paged out.

Although these events are highly unlikely, considering that the SDW must exist and be in main memory for the processor to observe that the PTW was faulted (modulo the associative memories), the page fault handler must be prepared to deal with these cases. The page fault handler needs the SDW for the segment to locate the page table for the segment and identify the particular PTW for the page on which the fault was taken, as only a segment number and computed address are supplied by the processor appending unit in the fault data.

The page fault handler depends upon non-local transfers by subroutines in the page-reading function; specifically, record quota overflow and out-of-physical-volume conditions in this function cause special action, including transfers back to the main path of the page fault handler.

The basic actions involved in handling a page fault are these:

1. Save all the processor machine conditions other than the SCU data, which was already saved. The page fault handler, unlike the segment fault handler, is the actual fault interceptor for this type of fault. Reset the processor mode register.

2. Mask to sys_level (it is not legal to accept interrupts during page control functions), and establish a stack frame on the base of the PRDS (processor data segment, wired per-processor stack) for the ALM page control environment (see "Stack Management," Section VIII).

3. Check for illegal conditions (page fault in ring 0 while the PRDS is in use as a stack) indicating system problems, crash if so.

4. Establish the ALM page control environment (initialize save stack, pointer register for the SST).

5. Try to lock the page table lock. Execute "Page Table Lock Waiting". Code if unsuccessful. (See "Page Table Lock Waiting," Section VIII.)

6. Perform the paging device housekeeping and replacement function, which ensures a small number of free PD records and currency of the PDMAP image on the bulk store (see Section VIII for details).

7. Determine whether this is a page fault taken on the descriptor segment when the processor needed on SDW (DSPTW APU cycle), a page fault taken pre-paging an EIS operand (PTW2 APU cycle), or a normal page fault on a page of a segment (PTW APU cycle). If none of the above, the processor is in error. In the first case, locate the page table for the descriptor segment of this process, and the page on which the fault was taken. The SDW for the descriptor segment is guaranteed to be in main memory. Proceed to step 9 in this case.

8. For a normal PTW or PTW2 cycle, try to obtain the SDW for the segment faulted on. If the descriptor segment is unpaged (as during initialization), there is no problem. Otherwise, check the PTW for the page of this process' descriptor segment containing the SDW for the segment on which the fault was taken. If this page is in main memory now, it can be read without taking a page fault. Pick up this SDW: a setfaults operation could have destroyed it at any time until this very instruction. If so, restart this page fault, abandoning the environment set up and unlocking the lock, causing the processor to take a segment fault by accessing that SDW. Otherwise, locate the page table and specific PTW for the page of the segment on which the fault was taken. (1)

9. Having located the PTW for the page on which the fault was taken, locate the AST entry for its segment.

(1) In the case of a PTW2 EIS prepage cycle, the computed address reported by the control unit in the SCU data must be adjusted one page up.

10. Check for two window situations involving some other process handling a page fault on that page before this process got the lock locked. In the first case the page is completely read in, and no page fault exists. In this case, unlock the lock, abandon the environment, and restart the machine conditions. The processor will then proceed to use that page. In the second case, the page could be being read in now, and is out-of-service on a read ("short page fault"). In this case, develop the wait event for the PTW and proceed to step 18.

11. Check for an error bit (ptw.er) set on by the interrupt side at the completion of a read from a previous page fault. If there was a read error, it turned on this bit (See "Error Strategy", Section VIII) and notified this process, so that it might take the fault over again and perform this step. Turn off the error bit, abandon the environment, unlock the lock, and transfer to the signaller (signaller$signaller) to cause the faulting process to signal page_fault_error.

12. Invoke the page-reading function (read_page) to find a main memory frame for the page and begin (and possibly complete) reading it in. This operation might possibly perform non-local exits in the case of record quota overflow (in which case that condition will be signalled in a manner identical to the signalling of page_fault_error in the previous step) or physical volume overflow (in which case the SDW will be faulted, the environment abandoned, the lock unlocked, and the machine conditions restarted to produce a segment fault). If the page-reading function encounters an RWS in progress on the page faulted on, set the abort bit ('pdme.abort) in the PDME for that page, causing the interrupt side to resolve the page fault (See "I/O Posting", Section VIII) with an "RWS abort", and proceed to step 18 to wait for this occurrence.

13. Meter this page fault. Compute the main memory usage charge of this process. Meter ring zero, directory, per-process, and level-one page faults. Compute the page-fault interarrival time histogram (displayed by print_paging_histogram) in the segment page_fault_histogram.

14. Execute the replacement algorithm write-behind function. This will cause writes to be queued, while the page read started by step 12 is in progress.

15. Now meter time spent processing this page fault.

16. If the page faulted on is not out-of-service, i.e., was either completely read in by step 12 or posted as complete by some actions occurring during step 14, the page fault is complete, and satisfied. Unlock the lock, abandon the environment, and restart the machine conditions. The process and processor will proceed to use that page.

17. The process must be made to wait for that page. If the page is involved in a bulk store transfer, "run" the bulk store (see "DIM Interface", Section VIII) until the page is no longer out-of-service, at which time go to step 16.

18. The process must be made to wait for a disk or RWS event. The page-reading function (or step 10) has developed the wait event. Transfer into the traffic controller environment as described in "Stack Management and Traffic Controller Interface" in Section VIII, causing the process to wait for this event, unlocking the lock, and abandoning the environment.

19. When such an event has occurred, or at least probably occurred, the traffic controller will transfer to page_fault$wait_return to restart the machine conditions. There is no page control environment or stack frame, and the page table lock is not locked. If indeed the interrupt side has posted this page, the process will resume and use the page. If indeed it has not (either the wakeup was spurious, as it is allowed to be, or the page has again been paged out in the window, the sequence of events starting at step 1 will be repeated.

SERVICES FOR SEGMENT CONTROL

Page control fills page tables and AST entries with information supplied by segment control, reports that information back to segment control as it changes dynamically, and performs operations upon those segments on behalf of segment control. The latter operations include truncating active segments, and evicting all of their pages from main memory and/or the paging device, so that segment control can deactivate the segments.

All of these functions, among others, are implemented in the PL/I program "pc". This program has available to it, via the transfer-vector "page", most of the functions in ALM page control described in the last section. Other than the activation-time service (fill_page_table), all of these operations are performed under the protection of the global page table lock. The program pc, as well as the other programs in call-side page control, use the entries pmut$lock_ptl and pmut$unlock_ptl to wire the current stack (3 pages of PDS), mask to sys_level, lock the page table lock, and undo all of these operations. In many cases, the entry page$cam is called before any unlocking (including that performed by the call-side wait coordinator) to make sure that any changes to PTWs are noticed by all of the system processors.

Activation-Time Service

(pc$fill_page_table)

This is the only fundamental service of page control that does not involve the page table lock. The entry pc$fill_page_table is called by segment control and the hierarchy salvager (among other parts of the system) to transform a file map in a VTOCE (see Section II) into a page table for use by page control. The routine is passed the AST entry pointer, the current length of the segment, and the PVT index to which the addresses in the file map refer. This routine does nothing more than translate the segment-control format addresses (see "Record Addresses" at the beginning of Section VI) and convert them into page-control format disk record addresses and null addresses, placing them in the PTW device address fields (mptw.devadd), initializing the rest of the PTWs as it goes. The PTW "first" bits, for the first-time PD performance optimization (See description of sst.ptw_first in Section VI) is initialized from that SST variable. A check is made, for each live address passed in, that it is indeed marked as "used" in the FSDCT (via a call to page$check_devadd). It is for this reason that the PVT index is passed as a parameter. This detects introduction of re-used device addresses into page control.

This service may be performed without the page table lock being locked. The caller guarantees that the segment whose page table is being filled is inaccessible, that no SDWs point at its page table, or will be made to point to it until after pc$fill_page_table returns. The check for reused addresses may also be made without the global lock locked; if the address is not reused, it will not be deposited in any possible window. If it is reused, it will stay that way whether or not the lock is locked.

File-map/Activation Attribute Reporting

(pc$get_file_map)

Segment control requires a reporting of the status of a segment and its addresses both at the time the segment is deactivated and at the time of the AST trickle. This information is used to update the VTOCE of the segment. The state of the addresses reported by this service to segment control is critical: it is a basic feature of the address management policy (see Section VII) that no nulled address ever be reported to a VTOCE file map. Thus a critical part of the file-map reporting service is the determination of whether or not an address should be reported to segment control at all. Part of the information returned to segment control is a list of nulled addresses that are to be deposited (returned to the free pool). The activation attributes of the segment are reported to the caller by filling in a complete copy of the AST entry for the segment, from a copy made under the protection of the page table lock. This copy must be made under the page table lock, in order for the "records used", "current length", and other fields to be consistent with themselves and with the list of addresses and list of addresses to be deposited that are returned.

Another action performed by the activation attribute reporting service is the maintenance of the "date-time-used" and "date-time-modified" fields in the AST entry. These fields are updated conditionally, depending upon the transparency attributes of the activations (see Section II), and the "file modified switch" (aste.fms).

All live addresses are reported to the VTOCE file map buffer passed in. Wholesale null addresses (representing no assignment of a record of disk) are also reported to this file map. The action taken for a nulled address depends upon several factors. A nulled address found in a PDME or CME (page on paging device or in main memory) must remain there; as long as a page has a frame of main memory or a PD record associated with it, it <u>must</u> have a disk record associated with it. A special <u>null</u> address (get_file_map_vt_null_addr, see "null_addresses.incl.pl1") is reported to the VTOCE for that page. The VTOCE will record no assignment for that page, as the nulledness of the address implied that the record of disk does not contain the page. A nulled address found in the PTW implies that the page has no main memory frame or paging device record associated with it (only disk addresses can be nulled). Normally, the action taken in this case is to report the address, not to the file map, but to a list of such addresses returned to the caller (the deposit list). The PTW is changed to contain a <u>null</u> address (get_file_map_vt_null_addr, again), and the caller is responsible for depositing all of the addresses in that list once it is known that the VTOCE has been successfully updated with the record address being deposited no longer in it. However, there is a class of circumstances in which the file-map reporting function may be inhibited from "culling" nulled addresses in this way. In these cases, nulled addresses in PTWs are left intact, and <u>not</u> reported to the deposit list of the caller. The caller may specify this behavior by passing the pointer to the deposit list as a null pointer. This action is also taken for segments with the switches aste.ehs and aste.dnzp both on. Such switches are set for hardcore segments in the normal AST used lists (and thus subject to AST trickle) which are prewithdrawn (such as the PDS of most processes). This action makes sure that prewithdrawn addresses stay withdrawn, i.e., are not deposited. See "Special Casing of Per-Process Hardcore Segments" in Section IV for motivation for this action.

The procedure pc$get_file_map is called with a pointer to the AST entry about which information is sought, a pointer to an ASTE image into which the AST information is to be copied, a pointer to a file map area in a VTOCE, into which the file map is to be placed, and a pointer to an array into which to put the deposit list. (As stated above, this pointer may be null). It returns, in addition to filling up the ASTE image, file map, and deposit list, a count of addresses put in the deposit list.

The procedure pc$get_file_map is also responsible for converting the page control format addresses into segment-control format (see Section VI), and turning off the bits aste.fms and aste.fmchanged (see Section II), indicating that any modification or file-map change for the segment has been noticed, and any further modification must be noticed independently.

Deactivation Service

(pc$cleanup)

At the time a segment is deactivated, any pages it may have in main memory or on the paging device must be evicted from these media. This must be done to satisfy the definition of a non-active segment, and to stabilize the state of the AST entry and file map.

The routine pc$cleanup is supplied a pointer to an AST entry for a segment to be so processed. The caller has ensured that no agency can bring pages of this segment in, either by having performed a "setfaults" operation on the segment, or being the only agency that has ever had access to the segment.

This routine is a prime example of routines that use ALM primitives and the multiplex wait protocol to process the pages of a segment in parallel, achieving state transitions by deterministic step.

With the page table lock locked, the following actions are performed for each page of the segment. The actions are repeated by reiterating over the segment until all pages of the segment are off the paging device, not undergoing RWS or paging I/O, and out of main memory. All addresses at that time will be in the PTWs.

1. Any page that is out-of-service (being read in or written out, perhaps by an earlier loop) has its wait event remembered, for potential waiting via the multiplex wait protocol.

2. Any page in main memory that is not out-of-service must be evicted; if it is modified (ptw.phm) the page-writing and purification function of ALM page control is invoked to purify it. If this puts it out-of-service, the PTW wait event is remembered for the multiplex wait protocol.

3. Any pure page is evicted by turning off its access bit and clearing the system caches and PTW associative memories via a call to page$cam_cache. If the page was modified in this window, restore the access and go back to step 2. The page then requires writing. If the page was successfully evicted, perform eviction cleanup (See Section VIII) not by a call to cleanup_page, but by inline PL/I code.

4. If the page still has a paging device record associated with it at this point (one may have actually been assigned in step 2, but this is rare), invoke the PD eviction subroutine (flush_one_pdrec) of pc to start an RWS, evict the page, or remember an RWS wait event as appropriate.

When all of the steps above have been performed for every page in the segment, and no steps (1, 2, or 4) remembered a wait event, the cleanup is complete.

Call-Side PD Eviction Subroutine

(flush_one_pdrec in PC)

This powerful subroutine is called by several services in the program pc, notably the deactivation service, PD reconfiguration service, truncation, post-crash-flush and shutdown services.

It is called with the variable "pdmep" pointing to a PDME describing a PD record that is to be vacated. The entries "flush_one_pdrec" and "delete_one_pdrec" differ only in the actions taken at the time the PD record is taken out of use; in one case the PDME is returned to the free list, and in the other case it is marked as deconfigured. Both of these entries evict the page from the paging device. The entry "truncate_one_pdrec", on the other hand, causes the destruction of the page, and the freeing of its PD record.

This subroutine, when not "truncating", starts an RWS for a PD record (via a call to page$pd_flush), which is modified with respect to disk, and not already undergoing RWS. It sets the multiplex wait variable "ind" to wait for any RWS that it starts (and does not find finished), or for any page out-of-service for paging I/O. For pages on the paging device not modified with respect to disk, it updates CMEs and PTWs, and frees or deconfigures the PDME.

One form of eviction from the paging device that is unique to this subroutine is that performed for pages in main memory (although not undergoing paging I/O). The paging device replacement algorithm does not evict pages from the paging device which have copies in main memory, because this is considered evidence of recent use. When eviction for such pages must be performed, however (as is the case in all call-side entries that need it), it is very simple to effect. The disk address from the PDME simply replaces the paging device address in the CME. The PDME "modified" bit (pdme.mod) is "or'ed" into the PTW "modified" bit (ptw.phm), using a key-line instruction. This causes the page to be written out to disk when it is evicted from main memory, in the case where the paging device contents were different from the copy of the page on disk, but the same as those in main memory.

This subroutine must not free PDMEs for PD records that have undergone RWS on behalf of the post-crash PD flush.

This subroutine sets "notify requested" bits in CMEs and PDMEs when it returns a wait event. This is superfluous, as the call-side wait coordinator will do this if such wait event is actually passed to it.

Truncation Service

The truncation of segments is performed for both segment control (from the Segment Control Truncation function (See Section IV)) and for supervisor subsystems that deal with non-hierarchy segments, in order to free their disk record resources.

Truncating a segment to length n (n given in pages) involves truncating all pages of page number equal to or greater than n. Truncating a page means associating zeros with the contents of that page; in fact, the actions performed to truncate a page in main memory are identical to those taken by the page-writing function (see Section VIII) when a page of zeros is discovered. Truncating a page which is neither on the paging device nor in main memory consists of no more than nulling its disk address (and updating the necessary ASTE quantities and quota cells). Recall that a nulled address is paged in by the page-reading functions as a page of zeros. Truncating a page on the paging device, whether or not it is in main memory, involves freeing the associated PD record by placing the PDME for it in the PD used list.

Truncating a segment consists of little more than performing the above actions, as is the case for each page. For pages that have paging I/O going on (ptw.os is on), the completion of this I/O is awaited via the multiplex wait protocol, For pages on the paging device, the call-side PD eviction subroutine (see preceding) is invoked (at the entry "truncate_one_pdrec"). Among the actions taken by this subroutine is the remembering of any RWS in progress on that page, for later waiting via the multiplex wait protocol. When all pages in the segment have been processed and no wait events remembered, the truncation is complete.

A large part of the complexity of the segment truncation primitive is the determination of whether or not a page being truncated was charged against quota. Basically, any page in main memory is charged against quota. Any page with a live disk address is charged against quota. Those pages with null addresses, or with nulled addresses but not in main memory, are not charged against quota.

At the end of the processing, the ASTE fields describing the number of records used and the current-length of the segment are updated. If quota checking is not inhibited for this segment (i.e., aste.nqsw is off), the quota utility quotaw$cu_for_pc is called to adjust the quota account against which the segment's record quota is charged. If any pages were actually truncated, the file-map-changed bit (aste.fmchanged) is set, indicating that segment control should update the VTOCE as soon as convenient, for addresses can be deposited, and must be removed from the VTOCE. Segment control's VTOCE update function will do both these things.

The page control truncation service does not require that no other agency in the system be creating pages while it is trying to truncate them. That is the problem of the subsystem or user code which invoked the storage system's truncation facility, not of page control. The only issue here is that the truncation service must be quite careful to multiply count pages it truncates multiple times. It is impossible for malice or accident to force the truncation service into a loop by so doing: only when the page table lock is unlocked while pc$truncate waits can such pages be created. The user cannot force paging-ins of nonzero pages in the truncated region, or their paging-out, or RWSs which are the only activities that will cause pc$truncate to wait.

The arguments to pc$truncate are the AST entry pointer of the segment and the length (n) to which it is to be truncated.

Boundsfault Service

(pc$move_page_table)

The segment control processing of a boundsfault usually involves the allocation of a new ASTE/page table pair for a segment, and the establishment of that ASTE and page table as the sole ASTE/page table for that segment. From the segment control side, the most critical operation here is the hashing of the old one out of the AST hash table, and the hashing of the new one in.

However, when such an operation is performed, if there are pages of the segment on which the boundsfault has taken place in main memory or on the paging device, there are page control data bases that describe the original ASTE and page table (PTWs) of the segment. Sometime during the processing of a boundsfault, page control must be invoked to construct a valid page table for the new ASTE, and modify all page control data bases that referenced the old ASTE/page table to reference the new one. This service is provided by pc$move_page_table, called with the two AST entry pointers involved.

It is also critical that all of the page control maintained data items in the ASTE be copied from old to new during the same locking of the page table lock (only one such locking will be required, there is no I/O involved) as that which the page table is reconstructed. This must be so in order that the interrupt side will reference the correct ASTE should any I/O on this segment complete, and so that any paging-out activity will do the same. The caller of pc$move_page_table (the boundsfault handler of segment control) ensures that neither ASTE is accessible, i.e., no process can access the segment on which the boundsfault has taken place.

The task of the page control boundsfault service is simple: all pages in main memory or on the paging device or on disk remain exactly as they are, whether or not I/O or RWSs are in progress on them. The essence of the task is to walk the old page table and new page table in parallel, chasing down any CMEs or PDMEs designated by the PTWs in the old page table, and changing the pointers in these CMEs and PDMEs to point to the new page table. The old PTW contents are copied to the new PTWs. The ASTE relative-pointers in the CMEs are similarly modified. The PTWs in the extent of the new page table beyond the extent of the old are similarly modified. The page table lock remains locked during the entire operation, ensuring that no process can use the data objects or change their state while they are being modified. Before the lock is unlocked, the entire ASTE contents (other than its threads and pool number) are copied from the old to the new ASTE.

One subtlety of the boundsfault service requires some clarification. The relative offset of PTWs into the SST segment is used as a wait event ID by processes awaiting the completion of non-RWS paging I/O (See "Wait Events", in Section VIII). When the page table has successfully been moved, the interrupt side will post any I/O which completes after that point by notifying the event ID associated with a new PTW. Thus, processes waiting for the page which began this waiting before the page table was moved are no longer waiting for the correct event, and will not be notified. Thus, the boundsfault service explicitly notifies any PTW event for which the CME notify-requested bit is on. This causes any process waiting for a PTW event associated with the old page table to run; when it successfully locks the page table lock, it will retry whatever it was doing, either via taking a segment fault or a page fault, and ultimately find the new PTW, and go to wait on that.

Modified-Switch Setting

(pc$updates)

Directories are normally activated with the transparent-modification attribute (see Section II for more illumination). This means that changes to the contents of the directory do not cause the file-modified switch of the directory to be set. This, in turn, means that the date-time-modified of a directory or its superiors is not advanced solely by modifying a directory. Although this convention dates from times when date-time-used was stored in a directory (it is now in the VTOCE for a segment) and change to this field had to be made without updating the date-time-modified of the directory, there is still a small class of operations (segment moving and online-salvaging) which modify directories in ways such that the directory date-time-modified should not be advanced.

The date-time-modified of a directory is defined recursively as the latest date-time-modified of any segment or directory under it, or such time that the directory was explicitly modified by directory control. In the case where segments are modified, the page control page-writing function notices the "modified" bit in the PTW (See "page-writing function" in Section VIII), and turns on the file-modified switch (aste.fms) in the ASTE of that segment and all of its superior directories (this bit is reported by the file-map and activation-attribute reporting service described earlier in this section). For the case of explicit modification of directories by directory control, an address-space management utility (sum$dirmod), to update certain fields of the directory. One of the actions taken by this program is to obtain an AST entry pointer for the modified directory via a call to "activate" (See "Significance of Activate," Section IV) and pass it to pc$updates. This entry, with the page table lock locked, does no more than chase up the ASTEs from that ASTE on up setting ASTE "fms" (file modified switch) bits explicitly.

POST-CRASH PD FLUSH

The management of the paging device is such that it contains information (copies of pages) that is not identical to copies of the same pages on disk. Records containing such pages are called "modified" PD records. In order to evict such pages from the paging device, a read-write sequence (RWS) must be performed. Part of the task of shutdown, normal or emergency, is to flush the paging device, i.e., evict all pages from it. This implies read-write sequences for all "modified" PD records. However, should a successful shutdown not be possible, the "modified" records of the paging device contain information duplicated nowhere else. The next bootload of Multics must copy the contents of these records back to the disk records to which they belong. This is known as repatriation of these pages. Repatriation of pages that had nulled disk addresses also involves resurrection of these addresses, implying modification of VTOCEs. The post-crash PD flush is the page control service that performs these tasks.

The paging device may be said to come in instances. An instance of a paging device is the paging device, its map, and all the pages which have ever been on it from the time that map is initialized, to the earliest of a successful shutdown, or flushing of the last record off of the paging device by post-crash flush, or abandonment by the operator "force_pd_abandon" ring-1 command (see the Multics Operators' Handbook, Order No. AM81). An instance of the paging device exists during only one bootload if that bootload shuts down successfully. Otherwise, it may exist during two or more bootloads. An instance of the paging device is uniquely identified by the "paging device time," the field pdmap_header.time_of_bootload, set to the clock time at which the paging device map was initialized. This field, along with the rest of the paging device map, is written out to the first few records of the bulk store every second by the PD housekeeping function in the page-fault handler. It is also written out by explicit calls to pc$write_pdmap.

An instance of a paging device that was created during the current bootload is said to be an active paging device; the system is said to have an active paging device. The bit fsdct.pd_active in the FSDCT header indicates this. An instance of a paging device that was created during some bootload other than the current bootload, and not successfully flushed (i.e., successful shutdown was not achieved) is called an unflushed paging device. When a hierarchy (or a bootload) has a paging device in this state, the system is said to have an unflushed paging device. The bit fsdct.pd_unflushed is on when this is the case.

Whenever a physical volume is accepted by a system with an active paging device, or an instance of a paging device is created (the paging device becomes active) during a bootload, the physical volume is said to have been exposed to that instance of the active paging device. Whenever a physical volume is accepted, the paging device time of the instance of the active paging device, if there is one, is written to the label of that volume before any segments on it are allowed to be activated. Whenever a paging device is made active during a bootload, a call is made (to the program fsout_vol, for each volume, see Section XIV) to write the paging device time to the labels of all volumes before that paging device is actually made available to the PD record allocator. Thus, any physical volume contains in its label the PD time of the last instance of the paging device to which it was exposed.

The label of the root physical volume (RPV) contains a bit (label.pd_active) which says whether it, and therefore the entire hierarchy which it commands, was exposed to an active paging device, this bit being cleared when the system is successfully shut down. If the system comes up after a crash and this bit is on, then the system must have an unflushed paging device, otherwise the system would have been successfully shut down and that bit cleared. Thus, the paging device map is read from the bulk store, and the PD time in the PDMAP header compared to that of the instance of the paging device to which the RPV was last exposed. If these are not the same, the paging device contents have been damaged (probably by the use of another hierarchy) since that RPV was last used (and not shut down). The system will not come up in this case; the operator must zero the paging device. If the system finds the paging device time on the bulk store zero, when the RPV was indeed exposed to an active paging device and not shutdown, it implies that the operator cleared it. A message is typed, and a new instance of the paging device is created. If the times indeed match, however, the system has an unflushed paging device, which must be flushed. The bit fsdct.pd_unflushed is turned on to this effect. All of the records of the paging device that are not "modified" have their PDMEs cleared (set to zero). Those marked as "deleted" by the PAGE CONFIG card (see "PD Reconfiguration" later in this section) are deleted. The state of the "modified" PDMEs left by these actions is regularized. There is no PD used list on an unflushed paging device. The map is written out in its new state. The records on the paging device will be repatriated as volumes are accepted. The manipulations described above are all performed in the program "init_pvt".

Whenever a physical volume is accepted by the system, it can tell whether or not it has been successfully demounted. Shutdown, it will be recalled, demounts all volumes (See Section XIV). Whenever a physical volume that has not been shut down is accepted, the physical volume salvager is invoked by volume management to salvage it. This physical volume salvaging, among other things, reconstructs the map of free addresses, and checks each VTOC entry (VTOCE) for consistency.

Whenever a physical volume that has not been successfully demounted is accepted by a system with an unflushed paging device, there exists the possibility that that volume was exposed to that instance of the paging device. If the volume was shut down successfully, it cannot have any records on any instance of the paging device. Only volumes present at the time of the crash can have records on this instance, the unflushed, current instance, of the paging device. Those volumes are exactly the set of volumes not successfully demounted which were exposed to this instance of the paging device. Whenever a volume that was not successfully demounted is accepted by a system with an unflushed paging device, comparing the PD time in the label of that volume to the PD time of the current, unflushed, paging device tells whether or not this is the case. Such a volume is said to have been <u>exposed</u> <u>to</u> <u>an</u> <u>unflushed</u> <u>PD</u>.

Whenever a volume exposed to an unflushed PD is being salvaged, records on that paging device will be repatriated to that volume. The task of identifying these records is facilitated by the recording of the physical volume table index of the volume containing the page in the PDMAP entry. This PVT index identifies the drive on which the volume was mounted during the bootload that crashed, which may not be the same as during the current bootload. However, the physical volume table index, as well as the PD time, is recorded in the pack label at the time the volume is accepted. Since the volume is guaranteed not to have been successfully demounted, it is impossible that any other volume could have had that PVT index after that during that bootload, and hence have pages on the unflushed paging device. Thus, by comparing the PVT index in the volume with that of each disk record stored as the "devadd" in a PDMAP entry, it can be determined precisely whether that PDME describes a record to be repatriated to this pack, and if so, to what disk address.

Repatriating the pages is only half of the task. Many of the "modified" pages on the paging device contain pages that were <u>never</u> written to disk; their entries in the file map of their segments' VTOCEs contain null addresses. Thus, simply writing back these pages to the disk is not enough, as a fault on that page will produce zeros, as the address in the VTOCE is null. Thus, in effect, such repatriations are resurrections; live addresses must be reported to the VTOCE. It is for this purpose that segment unique ID and page number are stored in the PDMAP entry. As each VTOCE (each segment) of a physical volume exposed to an unflushed paging device is processed by the physical volume salvager, a special service of page control (pc$flush_seg_old_pd) is called, passing the address of the file-map region of the VTOCE image, the old and current PVT indices of the physical volume, and the UID of the segment whose VTOCE is being processed as input parameters.

The entry pc$flush_seg_old_pd scans the entry PDMAP looking for PD records containing pages belonging to this segment; such entries have a matching UID, placed there by the PD record allocator (see Section VIII). For each such entry, an RWS is initiated by the use of the call-side PD eviction subroutine, flush_one_pdrec (see earlier description in this section). All of these RWSs are performed in parallel, and waited for in parallel via the multiplex wait protocol. A special bit (pdme.flushing) is turned on before each RWS is initiated, so that the interrupt side will neither clear nor free the PDME (see "I/O Posting", Section VIII). In order for the RWS mechanism to work, the PVT index in the PDME must be the PVT index of the volume in the current bootload, if the volume has moved. Thus, before invoking the call-side PD eviction subroutine, pc$flush_seg_old_pd saves the old PVT index in mpdme.save_old_pvtx, and places the new one in. This is done so that should the system crash during the processing of this segment, the next bootload can detect this (pdme.flushing will be on), and cleverly place the old PVT index back.

As the interrupt side completes each such RWS, it leaves the PDME intact. In all cases except the case of RWS (read or write) error, the disk address in the PDME will be a live disk address (RWS completion always causes a resurrection, if the disk address was nulled). However, a null address will have been left by the interrupt side if there was an error. In all cases except the last (RWS error), the live address from the PDME is moved to the appropriate slot in the VTOCE file map passed in as an argument (pdme.pageno says which slot), and two output parameters, representing segment current length and "records used", are adjusted if a resurrection took place. The PDME is then zeroed, but not freed.

The post-crash PD flush repatriation procedure substantially increases the time required to do a physical volume salvage, as the whole PD map must be scanned for each VTOCE processed.

The placing of a UID in a PDME ensures that there are no windows between the last time the map was written out and the last time data was written to that PD record. Were this not the case, the wrong data might be flushed to some segment.

After all VTOCEs have been processed by the physical volume salvager, a special primitive (pc$cleanout_old_pd_pv) is called to clear PDMAP entries for "parasite" segments (i.e., those with no VTOCEs), such as descriptor segments on the RPV (see Section VII). This primitive also checks that no records exist on the paging device which belonged to the volume being salvaged; they should all have been repatriated. If there are any, very little can be said or done about them, and nothing would be gained by crashing the system. An informative message about the curiosity is typed out.

SHUTDOWN AND DEMOUNTING SERVICES

The aims of both shutdown and demounting are to ensure that the paging device and main memory contain no pages of a set of physical volumes; in the case of demount, it is one physical volume. In the case of shutdown, it is all of the volumes present. Demounting causes this to occur by deactivating all of the segments on the volume. Shutdown, however, although it goes through the demount procedures for all volumes present, does not attempt these deactivations.

Shutdown flushes the paging device (evicts all pages on it) as early as possible. This is so that the system should not have an unflushed paging device, should shutdown fail. Obviously, only active paging devices can be flushed. The entry pc$pd_flush_all exists for this purpose. It calls the call-side PD eviction subroutine flush_one_pdrec (see earlier description in this section) to flush each page off of the paging device (initiating RWSs if modified), and uses the multiplex wait protocol to multiplex the wait events. When this routine returns, the paging device may be declared inactive.

Shutdown also flushes all of main memory before doing its update_vtoce loop; this is so that any disk record addresses for pages in main memory (the paging device has been flushed, as above, at this time) are resurrected prior to the VTOCE updates. The routine pc$flush is called for this purpose. It calls the page writing/purification function in ALM page control (See description in Section VIII) to initiate writes on all pages that are modified with respect to main memory. All I/Os are awaited (whether or not this action started them) via the multiplex wait protocol. This action also causes all pages of zeros in main memory to be evicted, nulling their addresses.

Normal and emergency shutdown call the primitive pc$write_pdmap several times to write out the PD map to the bulk store when significant changes to its state are made. Writes performed by this primitive are done via using the segment pdmap_seg as an abs-seg over the bulk store; such writes are done via calls on the page-writing function in ALM page control, and are posted normally via the interrupt side. This does not conflict with writes to the map performed by the PD housekeeping function, which may even be going on simultaneously; these writes do not involve PTWs or CMEs, and will not even be reported by the bulk store DIM to the interrupt side upon completion. The function is also used by PD Reconfiguration, see Section X.

Volume demounting does not require any special services from page control; all of the flushing of pages out of main memory and off of the paging device are performed by pc$cleanup, invoked by segment control when segments on that volume are deactivated (see Section IV). However, a special entry in page control is called by the volume demounting function after all segments have supposedly been deactivated. This entry, pc$check_pd_demount, does no more than check that no pages belonging to that volume are still on the paging device. This is solely as a check for bugs; it should never be the case that there are such records.

RECORD ADDRESS DEPOSITING SERVICES

pc$deposit_list
pc$list_deposited_add
pc$truncate_deposit_all

Page control, as the maintainer of the FSDCT bit maps for mounted volumes, is charged with the depositing of addresses on behalf of segment control and other agencies.

The entry pc$deposit_list is called with a "deposit list", an array of addresses to be deposited, and a PVT index identifying their volume. Such a "deposit list" is produced by the file-map reporting service (pc$get_file_map), and by the segment control segment truncation facility in the program truncate_vtoce. The number of entries in this array is also a parameter. Basically, this entry does nothing more than iterate over the array supplied and call the withdraw/deposit mechanism in the program free store (See "Individual Mechanisms" in Section VIII) with each address and the PVT index. This operation is performed without the protection of the page table lock: depositing is a unitary operation that involves no races, as only one process can deposit a given address at one time.

The entry pc$list_deposited_add is an entry that performs that function of the file-map reporting service which is the reporting of nulled addresses and their replacement in PTWs by null addresses. This entry places the addresses so gathered into a "deposit list", such as that accepted by pc$list_deposit above. This operation must be performed under the protection of the page table lock. The criteria for reporting an address are the same as those in pc$get_file_map, i.e., the address must be nulled and in a PTW. Addresses so reported are replaced in the PTW by the null address "list_deposit_null_addr" (See null_addresses.incl.pl1). This entry is used by code dealing with non-hierarchy segments, such as some initialization code, and by the segment mover (See "Segment Moving" in Section IV).

The entry pc$truncate_deposit_all is a macro operation consisting of successive calls to pc$truncate (to zero length), pc$list_deposited_add, and pc$deposit_list. It is used to destroy RPV parasite segments (e.g., PRDSs and descriptor segments). It is supplied an AST entry pointer as an argument. There is no window between the truncation and the depositing: these segments have no VTOCEs, and are not under consideration by any subsequent bootload.

PAGING DEVICE RECORD DELETION

The paging device reconfiguration service is described in Section X, "Peripheral Services of Page Control", as it does not interact with the mainstream of page control, and in general only deals with deconfigured paging device records. However, one part of the paging device reconfiguration software involves taking paging device records out of use. This involves the use of the services of the kernel of page control.

All paging device reconfiguration is managed by the program delete_pd_records, which wires itself via the wire_proc mechanism (See Section X) when invoked. In all cases except the use of the "delpage" operator command, it deals with unflushed paging devices and deconfigured records. It does not involve PTWs, CMEs, or pages of segments.

However, the "delpage" operator command may involve the eviction of pages from in-use paging device records. In this case, delete_pd_records, which is at that point running masked and wired with the page-table lock locked, invokes the entry pc$delete_pd_records with the first index and number of paging device records to be deleted. Using the call-side PD eviction subroutine, flush_one_pdrec (called at the delete_one_pdrec entry) (See earlier description of this subroutine), this routine evicts pages, starting RWSs where necessary, and waiting for all paging and RWS I/O via the multiplex wait protocol.

The eviction of pages from PD records performed for record deletion is different from those evictions performed for deactivation, truncation, or the PD replacement algorithm, etc., insofar as the PDME for the PD record is not to be threaded into the PD used list, but left out, with a word of all ones set in its thread word. This marks the "deconfigured" state of the paging device record. When a nonmodified page is evicted by delete_one_pdrec, this subroutine performs this deletion. Otherwise, the bit pdme.removing is set on before or during RWS, so that the interrupt side (See "I/O posting" in Section VIII) will deconfigure the record instead of threading the PDME into the used list.

Paging device records are also deleted automatically by the interrupt side when paging device I/O errors have occurred; see "Error Strategy" in Section VIII.

FORCED SEGMENT I/O AND WIRING

(pc_wired)

Several agencies in the system have the need to "wire" portions of segments (make their pages nonremovable from main memory). In some cases, this is accomplished by turning on "wired" bits in the PTWs for the affected pages and simply touching them. This technique is generally used to wire regions of stacks. A less ad-hoc facility is available though, through the entries pc_wired$wire, pc_wired$wire_wait, and pc_wired$unwire. Typical uses of this facility are for wiring data bases to be used at interrupt time by facilities that are not always enabled (such as the ARPANET software), and by wire_proc, the manager of procedure-wiring requests (described in Section X), which temporarily wires procedures that must not take page faults.

The program pc_wired implements all of these functions, along with a few others described below. In all cases, it is passed the AST entry pointer for some segment (it is the caller's responsibility to ensure that the segment is either a supervisor segment or cannot be deactivated while pc_wired is operating upon it), a first page number, and a number of pages to be read/written/wired/unwired. In all cases, a number of pages of -1 indicates that all pages from the "first page" specified to the end of the segment are to be read/written/wired/unwired.

The service provided by pc_wired$wire_wait is the most often used. It wires the pages of the segment specified, if they are not already wired, and does not return until they are all in main memory. It operates by turning on all of the "wired" bits in the affected PTWs that are not already on, and initiates reads via calls to the page-reading function (See Section VIII) via the transfer-vector page$pread. All I/O operations on these pages, whether noticed or started by this module or reported back by page$pread (which may include FSDCT pagings, RWS events, etc.) are awaited via the multiplex wait protocol, until all specified pages are in main memory, with no I/O in progress on them.

The service provided by pc_wired$wire is similar, but it does not retry calls to ALM page control or wait for I/O completions. Thus, its effect is little more than to turn on all of the wired bits involved and start some of the I/Os. This service is not particularly useful, and is not used.

The service provided by pc_wired$unwire is commonly used. It simply unwires the pages wired by either of the two above entries. In all cases, this is nothing more than turning off the PTW "wired" bits.

The module pc_wired also provides a set of services to perform paging I/O on demand upon segments. These are used by the physical volume salvager to pre-page (i.e., start asynchronous paging-in) segments used to address VTOCs, and by the segment mover to force zero pages to be noticed (and thus have their addresses nulled) by the page-writing function (See Section VIII). A special form of this service is available to the directory control directory unlocking primitive, via pc_wired$write_wait_uid. This entry is used when a directory is unlocked which directory control knows to have been modified; it causes all the modified pages to be written from main memory (perhaps to the paging device) as a hedge against crashing. It is different from the service provided by pc_wired$write_wait in that the caller makes no guarantee that the AST entry pointer provided will remain valid during the operation of pc_wired; therefore, a segment unique ID (UID) is supplied by the caller so that pc_wired can check the AST entry each time the page table lock is relocked, to ensure that it still designates the same segment.

The entry pc_wired$write_wait is the most generally used. Given that the caller ensures that no process may be modifying the segment, it ensures that no modified pages of the segment (in the range specified) exist in main memory when it returns. The entry pc_wired$write performs similar actions, but does not wait, and makes no statement about the state of the segment when it returns, and thus is not used.

The entry pc_wired$read is used to start page-reads for all pages in the range specified. It makes no guarantees about when these reads will be complete; this is used solely as a performance optimization feature, for those supervisor subsystems that can anticipate their page reference patterns. There is no entry pc_wired$read_wait; if there were, it could not possibly guarantee that pages which it had read would stay in main memory when it returned, for any paging activity at all could evict them. The concept of reading pages in nonevictably is the concept of wiring, treated above.

The entries pc_wired$read, pc_wired$write, and pc_wired$write_wait all operate by calling the page-reading and page-writing functions in ALM page control, and iterating in the "wait" cases via the multiplex wait protocol.

ABS-WIRING SERVICE

(pc_contig)

Peripheral device operation via the IOM (Input-Output Multiplexer) requires contiguous regions of main memory for data buffers. The IOM provides a facility whereby arbitrary user-supplied channel programs may be run in a given region of main memory, preventing them from damaging other regions of main memory via a per-channel limit register. The same facility also relocates addresses appearing in such channel programs with respect to the base of the region, such that the writer of such a channel program need not even know where in main memory the channel program (and the data) will appear. The use of this facility is managed by the I/O interfacer.

A critical part of this facility is the ability to acquire successive page frames of main memory that can be made to form a contiguous region. When storage system segments are to be used as buffers for the IOM, they must be paged into such regions of pages, in address order, and not be evicted from those page frames for any reason, including deconfiguration of memory. Such pages may not be moved around main memory, as is done by some of the functions described in Section VIII. Such pages are said to be "abs-wired", as are the segments to which they belong at the time that their pages are in this state.

The use of abs-wired buffers, for the IOM (and the FNP6600 Communications Processor bootload software, through the IOM) is managed by a program called the I/O buffer manager (iobm). This program calls the page control segment abs-wiring service to allocate regions of main memory and abs-wire segments into them. It uses timers and request queues to schedule re-use and unwiring of these buffers. The I/O buffer manager also performs the unwiring of these buffers when that act is appropriate; it turns off CME abs-wired bits and PTW wired bits, operations that need not be protected by the page-table lock.

The program pc_contig is responsible for abs-wiring portions of segments. To abs-wire a portion of a segment, a number of usable main memory frames equal to the number of pages of the segment to be abs-wired must be found. A main memory frame is usable if it is in a non-deconfigurable system controller, is not deconfigured already, is not already in use by an abs-wired segment, and is in the first 256K of main memory. All frames found must be in the same system controller. (The issue of the first 256K involves an IOM design issue known as "backup list service".

The entry pc_contig$wire is called with the AST entry pointer of the segment whose pages are to be abs-wired, the number of the first such page, and the number of such pages. It returns a core map entry pointer to the first core map entry of the region into which it allocated and paged in and abs-wired the pages of the segment, from which the I/O buffer manager computes the main memory address of the region. This pointer is returned as null if the requested allocation could not be performed. A flag is also passed indicating whether or not this entry has been called on the interrupt side; currently, it never is.

The entry pc_contig$wire locks the page table lock and scans the core map for a sufficient number of usable main memory frames; if there are not enough, it tries several times to call the I/O buffer manager to release any frames it possibly can which it is holding. Only if this repeatedly fails is the caller informed that the requested allocation cannot be performed.

Once a region of main memory is decided upon, all pages currently residing there are evicted via the demand eviction function described in Section VIII. All I/O and RWSs in these frames are waited out. The pages of the segment to be abs-wired are read in via the page abs-wiring function described in Section VIII. For each page, this reading does not commence until the previous contents of the frame have been evicted, and I/Os already in progress there waited out. All of these operations are paralleled and waited for in parallel via the multiplex wait protocol.

Another interface to the abs-wiring service is maintained for historical reasons in the program pc_abs, at the entries pc_abs$wire_abs and pc_abs$unwire_abs. These entries are called with AST entry pointers, number of pages to be wired or unwired, and the number of the first such page. In the wiring case, this entry does nothing more than call pc_contig$wire. In the unwire case, the CME "abs-wired" bits and the PTW "wired" bits are turned off, again not requiring the protection of the page-table lock. This set of interfaces is currently used only by the ARPANET software to create buffers for the Interface Message Processor (IMP).

MAIN MEMORY DECONFIGURATION SERVICE

The Multics Dynamic Reconfiguration Software (See the Multics Reconfiguration PLM, Order No. AN71) provides the ability to take single frames of main memory out of use, and to take many out of use in order to take an entire system controller out of use. The commands which perform these activities are the "delmain" and "delmem" commands (see the Multics Operators' Handbook, Order No. AM81). Taking frames out of use in this way is performed by the page control main memory deconfiguration service provided by the entry pc_abs$remove_core. The entire power of this program is derived from the demand eviction function described in Section VIII and the multiplex wait protocol.

The program pc_abs, when invoked at this entry, with the first frame number and number of frames to be deleted, starts off by making legitimacy checks; the system must be left with enough main memory to function, and no frame that contains an abs-wired page may be deleted. The program wires itself via the procedure temp-wiring service described in Section X, and locks the page table lock, masking and wiring its stack via pmut$lock_ptl (see Section VIII, "Lock Conventions").

The program iterates over the region to be deleted, assured of the legitimacy of the request, turning on the "removing" bit (cme.removing) in the core map entry for each main memory frame to be deleted. This ensures that the main memory frame allocator (find_core) will never allow this frame to be allocated to a page; this ensures the deterministic success of the eviction that follows. The demand page eviction function is invoked on each frame that is not already deleted, until all frames are deleted. The wait events returned by page$evict are multiplexed by the multiplex wait protocol. Each frame from which a page has been evicted, with no wait event, is threaded out of the main memory used list, and given a thread word of -1.

The program returns, unlocking the page table lock, unwiring its stack, and unwiring itself.

SERVICES FOR TRAFFIC CONTROL

Traffic control performs many services for page control, notably implementing the wait/notify mechanism by which the waiting for of many page control events occurs. Page control also performs three services for traffic control: the loading and unloading of processes, and the post-purging of a process.

Process Loading

(wired_plm$load)

The two critical pages of a process (the first page of the descriptor segment and the process data segment (PDS)) must be wired before a processor is allowed to run in that process: A process in this state is known as loaded. The loading of a process is performed at the time it acquires eligibility. The loading of processes is performed by the program wired_plm, which has as its sole entry point the entry wired_plm$load.

The process-loading function is different from any other service in page control insofar as it performs its task on behalf of some other process than the one in which it is invoked. The process-loading function is invoked from the traffic controller's "getwork" routine (with the traffic control lock unlocked) at the time a process is being made eligible. Since loading a process may involve waiting for the reading (or RWS completion) of the two critical pages, waiting must be performed if this is the case. The process that is currently running in the traffic controller cannot and should not be made to wait for these events, involved in the loading of some arbitrary process. Thus, the process that is being loaded is made to wait for the events involved in its own loading itself. The traffic controller will not try to run any process that is waiting for an event, whether or not that process is loaded. When an event is notified, the traffic controller will usually try to run a process that had been waiting for that event. However, if that process is not loaded, a call will be made to the page control process-loading function to achieve or continue the loading of that process. Only when the loading function returns the result that the loading of the process is complete does the traffic controller mark it as "loaded", begin to run it, and interpret notifications of wait events in the normal way.

Thus, when a process is made eligible (it is never loaded at the time it is made eligible) a call is made to the loading function to start as many operations as can be started in parallel to accomplish its loading. If the process-loading function returns the fact that the process is loaded upon return from this function, then that is the case. Otherwise, the process-loading function returns a wait event of some operation that it started which was not completed. Since the traffic controller will cause the process being loaded to wait for that event, and call the loading function back when that process is notified, the effect is that of the process-loading function being called back when that event has been notified. Thus, in effect, the process loading function is called in a loop for each loading, returning either a wait event or an indication of loading having been successfully achieved each time. It is called again and again each time it returns a wait event, after that event has occurred, until it returns an indication of complete loading. This is a strategy very much parallel to the simple and multiplex wait protocols used elsewhere in page control.

The loading function is invoked without the traffic controller lock locked. It locks the page table lock, and unlocks it when done. Since the traffic controller locks its lock, upon return from the process-loading function after the latter has unlocked the page table lock, there is a window between these two events during which the event handed back by the process-loading function might occur. Thus, the traffic controller "validates" such events by actually checking PTWs and PDMEs designated by such events for valid out-of-service or RWS states (See Section VIII for further discussion of this anomaly). If such an event is found to be "invalid", the process being loaded is set up so that the process-loading function will be called again for it as soon as possible, i.e., the process-loading function will be retried without any wait.

The code of the process-loading function itself is very simple: it develops wait event IDs for either of the two critical pages it finds out-of-service or undergoing RWS, and invokes the page-reading function of ALM page control (page$pread) to read in either of the two pages not in main memory, remembering the event ID of any event detected by this primitive. It returns to the traffic controller any of the wait events encountered in either of these ways; if there are none (both pages are in main memory), it returns to the traffic controller the fact that the process is loaded. This code also turns on the "wired" bits in the PTWs of the two critical pages if they are not already on; this is part of its contract, and ensures completion of the read operations in a deterministic number of steps.

Process Unloading

Process unloading consists solely of turning off the "wired" bits in the PTWs for a process' two critical pages. This operation, which need not be performed under the page table lock, is done by the routine "unload_old_process" in page fault, invoked solely by the traffic controller and returning to it.

Post-Purging

The post-purging service of page control is used as a performance optimizing algorithm to bias the page replacement algorithm in favor of replacing pages of a process that loses eligibility. This service is invoked by the traffic controller at the time that a process loses eligibility, for any process whose work class indicates that post-purging is to be performed. Part of the post-purging service also consists of estimating the "working set" of the process, used by the traffic controller in the decision to grant eligibility.

The basic task of the post-purging function is to scan the per-process trace list of pagings performed by a process (see Section VIII for the "per process page trace list") (this function runs in the process it is processing), and to classify the pages involved in the various paging-ins as part of the process' working set or not, and bias the page replacement algorithm in favor of their replacement in various ways.

The post-purge function is implemented in the ALM program post_purge, called by the traffic controller with the traffic control lock not locked. It locks the page table lock at the start of its processing, and unlocks it only at the end of its processing, before returning to the traffic controller.

The post-purge function considers each page reading in the trace list, between the last time the process was post-purged and the current time. It also makes an entry in the trace list, a "scheduling" entry, for use by the "page_trace" command. It considers six attributes of each page in the trace list, and performs up to four potential actions for each page based upon them. These attributes are:

1. The page being in main memory at the time it is seen.

2. The page being on the paging device.

3. The page being part of a per-process (aste.per_process on. segment.

4. The page having its "used bit" (ptw.phu) on, indicating recent use.

5. The page having its "used in quantum" bit (ptw.phu1) bit on, indicating use since the last post-purging if certain options below are selected.

6. The page having its "modified" bit on.

The four actions that can be taken for each page are:

1. Call the page_writing function to write the page out.

2. Move the main-memory page frame for the page in the used list to the least-recently-used (most replaceable) position.

3. Turn off the used and used-in-quantum (ptw.phm and ptw.phm1. bits.

4. Count the bit in the working set of the process.

The mapping from all sixty-four possible combinations of these attributes into any sub set of the four possible actions is determined by the table "code_tree" in this program. The actions specified by this table in release 5.0 are:

1. Call the page-writing function to write the page out. This is <u>NEVER</u> selected.

2. Move the page to the least-recently-used position of the main memory used list. Done for any page meeting criteria 1 and 3, i.e., in main memory and part of a per-process segment.

3. Turn off the "used" and "used in quantum" bits. This is <u>NEVER</u> selected.

4. Count the page in the process' working set. Done for pages meeting criterion (4), i.e., the used bit is on.

There are no installed tools to change the contents of this table, or interpret them.

The post-purge function marks the bit ptw.processed (also known as ptw.er or ptw.pre_paged) in every PTW it processes; it turns off all bits it so turned on before it finishes. Any page it finds with this bit on must already have been processed by this pass; such occurrences are considered evidence of "thrashing", and are counted in the meter sst.thrashing. They indicate the occurrence of a process not being able to keep a page it was using in main memory during one period of eligibility. This action might also turn off PTW error flags by virtue of sharing of this bit, but the worst effect of this would be to cause a process to take an extra page fault to retry and perhaps rediscover a disk or paging device read error.

SECTION X

PERIPHERAL SERVICES OF PAGE CONTROL

This section covers three mechanisms used by the supervisor that can be construed as being part of page control. These three mechanisms are:

1. The facility that temp-wires procedures and their linkage.
2. Paging device reconfiguration.
3. Main memory frame freeing.

These mechanisms do not directly deal with page control objects that are in use. In the first case, no page control objects are dealt with at all; all manipulation of pages is performed by calls upon the services described in Section IX. In the second and third case, objects (CMEs and PDMEs) are threaded into their respective used lists, under the protection of the page table lock. Part of paging device reconfiguration is involved with taking PD records that are in use out of use; this is performed by a page control service described in the previous section ("Paging Device Record Deletion").

PROCEDURE WIRING

(wire_proc)

Many procedures in the Multics supervisor are wired, i.e., may not be removed from main memory. Often this is on account of the fact that they are used during page or traffic control operations, or in processing interrupts. Code invoked in such circumstances may not take page faults, for the taking of page faults may involve page control or traffic recursively, or cause the processor to be lost while a per-processor resource is in use.

Some procedures that may not take page faults are not invoked often; such procedures include much of the code that implements the reconfiguration, and much of the code of the ARPANET interface. Such procedures cause themselves to become wired when they are invoked, and unwired when they return. This procedure wiring/unwiring function is performed by the program wire_proc.

The program wire_proc does not deal with page control data bases at all; it calls pc_wired$wire_wait and pc_wired$unwire (described under "Forced Segment I/O and Wiring," Section IX) to wire and unwire the segments and parts of segments it deals with. The program wire_proc is not itself wired, and does not deal with the page control environment. The basic task of this program is to multiplex requests to wire the same segment; a table is kept of segments it has wired, in the region "sst.wire_proc_data" in the SST. When a request is made to wire a procedure, a check is made to see if that procedure has already been wired by this mechanism, in which case an entry in this table exists for that segment. If not, an entry is made for the segment, and a call to pc_wired is made to wire the segment. In any case, a counter of processes that have called to wire that segment, kept in the table entry, is incremented. When a process calls to unwire the segment, the counter in the table entry (which must exist) is decremented. If and only if the counter reaches zero, a call is made to pc_wired to unwire the segment. Thus, the segment remains wired from the time the first process calls to wire it until the last process calls to unwire it.

Whenever wire_proc wires or unwires a segment, the region of the appropriate supervisor linkage section that contains the linkage for that segment (if it has any), is wired or unwired as well. The program wire_proc checks that it does not try to wire portions of unpaged segments: this case may occur during initialization, when procedures that call wire_proc, which later become paged (See the Multics Initialization PLM, Order No. AN70) are still unpaged, and in cases of procedures with the "wired" attribute defined for their linkage sections in the MST Header (See "generate_mst" in the System Tools PLM, Order No. AZ03, and AN70).

The operations of the program wire_proc, and its table in the SST (which is defined in wire_proc_data.incl.pl1) are protected by a lock, the cell wpd.temp_w.lock in sst.wire_proc_data. Since wire_proc is used by system initialization in collection 1, before the system locking facility is available, wire_proc locks and unlocks this lock, and waits for its unlocking via explicit calls to "stac" and "stacq", with calls to pxss$addevent, pxss$delevent, and pxss$notify for synchronization. The value of the event ID for the unlocking of this lock is "200000000000"b3, and is stored (by init_sst) in the cell sst.temp_w_event in the SST.

The program wire_proc has two sets of entries, wire_proc/unwire_proc, and wire_me/unwire_me. The first two are very rarely used; their caller provides a pointer to the segment to be wired or unwired. The latter are the common pair; the program (i.e. the segment) that calls these entries is assumed to be the target of the wiring or unwiring, and is wired or unwired accordingly.

PAGING DEVICE RECONFIGURATION

(delete_pd_records)

(See also the discussion in the Multics Reconfiguration PLM, Order No. AN71.)

The storage system provides the ability to remove records of the bulk store paging device from use, and add them back. This facility is made available through the operator "addpage" and "delpage" commands. It is implemented in the procedure "delete_pd_records", a part of system reconfiguration that wires itself (via the procedure-wiring service described earlier in this section) and locks the page table lock (via pmut$lock_ptl, see Section VIII) when it actually deals with in-use page control data bases. This procedure also has several entries called by the ring-1 operator environment (See the Multics Operators' Handbook, Order No. AM81) to deal with unflushed paging devices (See "Post Crash PD Flush", Section IX). Among these are included entries that allow the number of unflushed records to be determined, the unflushed instance of the paging device to be abandoned, and a new instance of the paging device to be created and made active.

Except for the part of record deletion that involves evicting pages occupying PD records to be deleted, none of these operations involve dealing with PD records or their PDMEs that are actually in use. Even the operations that free records simply make them available for use. The operation of evicting pages from regions of the paging device being taken out of use is performed by the entry pc$delete_pd_records, which utilizes the methods and routines of the page control kernel to accomplish this. Thus, the procedure delete_pd_records never concerns itself with PTWs, CMEs, or pages of segments.

The entries that add and delete PD records (add_pd_records, delete_pd_records) are called with the first number and number of PD records to be added/deleted. They wire themselves and lock the page table lock when inspecting the paging device map. They both check the validity of their arguments before so doing. The entry to delete PD records does nothing more (once wired, masked and locked) than call pc$delete_pd_records to delete the records; the entry to add PD records does nothing more than thread entries in the region to be added, clearing them and checking before so doing that they were in fact deconfigured previously (first word = -1). Note that entries can be deconfigured by initialization (the program init_pvt) as well as delete_pd_records. Both procedures invoke a subroutine (check_pd_free_and_using) to scan the changed PD map and compute from scratch the parameters sst.pd_free and sst.pd_using, and update the PDMAP header (see Section VI). They call pc$write_pdmap (see Section IX, "Services for Shutdown/Demounting") to write out the changed map to the bulk store. These routines also change the actual "PAGE" CONFIG card in the Multics configuration deck image to indicate up to five pairs of deleted regions of the paging device. If there are more than five deletions, the non-fatal syserr message "delete_pd_records: page card cannot be generated" is issued, and only the first five placed on the card. The PAGE card image, created by the subroutine "build_page_card", is provided only for the use of the "print_config_deck" (pcd) command. The entries add_pd_records may not be used if an unflushed paging device exists.

The four entries scrap_entire_pd, check_pd, scrap_pd_recs, and enable_pd are for use of the ring-1 initializer operator environment for dealing with unflushed paging devices. None of them deal with active paging devices, and thus, they do not wire the procedure delete_pd_records, or lock the page table lock or mask.

The entry delete_pd_records$scrap_entire_pd is invoked by the ring 1 "force_pd_abandon" command. It scans the PD map of an unflushed paging device for any records still in use, (i.e., unflushed, containing unrepatriated pages). As long as such records exist, the system cannot be brought out of ring 1. This entry marks these records as no longer in use, thereby acknowledging that their repatriation has been deemed to be impossible. This operator command is used when a physical volume has been destroyed, and repatriation of PD records to it has become impossible.

The entry delete_pd_records$scrap_pd_recs is similar to delete_pd_records$scrap_entire_pd, but only the PD records pertaining to one physical volume are "scrapped". This facility is not currently used.

The entry delete_pd_records$check_pd is used by the ring 1 operator environment to determine if there are unflushed (unrepatriated) paging device records on an unflushed paging device. It returns the number of such records, and the number of physical volumes on which they appear (the count of distinct PVT indices in the PDMAP entries). Only if there are no such records may the system be brought up. Such records may be taken out of this state by either repatriation (via accepting the physical volumes from whence they came) or the "force_pd_abandon" command, which scraps them.

The entry delete_pd_records$enable_pd is called by the ring-1 operator environment to initialize a new instance of the paging device and its map. It is called at the time the system leaves ring 1, which can only happen if there are no unflushed records on an unflushed paging device. This facility is only used in the case of an unflushed paging device; its first step is to check that this is the case, and in fact that there are no unflushed records (via a scan of the map). This entry scans the map, zeroing all PDMAP entries that are not marked (by init_pvt) as deconfigured, and threads them into the PD used list (as free entries). This action does not allow them to be used; only the variable sst.pd_using being set to a nonzero value enables the PD allocator. Thus, this threading need not even be performed under the page table lock. When all of these entries have been threaded in, the clock is read which produces the unique time value that identifies the instance of the paging device being created, which will be used at post-crash PD flush time (see Section IX) should the system crash non-recoverably with this instance of the paging device active. The subroutine check_pd_free_and_using, described above under the description of add_pd_records and delete_pd_records, is invoked to set the SST variables sst.pd_free and sst.pd_using, and copy relevant parameters into the PDMAP header. The setting of sst.pd_using actually puts the paging device into use, and enables the PD allocator. The labels of all mounted physical volumes are written out, via calls to fsout_vol (see Section IV). This causes the fact that they were exposed to the new instance of the paging device to be recorded in their labels, for possible later use by the post-crash PD flush. As a final action, the active map is written out to the bulk store (via pc$write_pdmap), and a syserr message issued.

MAIN MEMORY FRAME FREEING

Initialization adds page frames of main memory to the paging pool (i.e., removes them from the deconfigured state in which init_sst creates them) as initialization progresses. Similarly, system reconfiguration adds page frames to the paging pool on behalf of the operator "addmem" and "addmain" commands. This facility is provided by the program freecore, which wires itself (via the procedure-wiring facility described earlier in this section) and masks and locks the page table lock (via pmut$lock_ptl, see Section VIII) when invoked. This procedure never deals with main memory frames that are actually in use; thus, it never deals with PTWs, PDMEs, or pages of segments. It is called with the main memory address (as a page frame index into main memory) of a page frame to be freed; it checks, under the page table lock, that indeed, that page frame is deconfigured before proceeding any further.

The program freecore checks the page frame that is to be added for parity errors (via a call to pmut$check_parity_for_use) prior to adding it, typing a syserr message if a parity error occurred. Otherwise, the main memory frame's CME is threaded into the main memory used list, starting this list if it is the first frame so added. Various CME flags and fields are cleared at this time, and the pointers sst.usedp and sst.wusedp (See "Main Memory Replacement Algorithm," Section V) are set to point to this page frame's CME. Counters and meters in the SST and SCS are updated as well.

SECTION XI

QUOTA MANAGEMENT

Quota (page quota, record quota) is the mechanism by which the consumption of segment storage space is administratively controlled. Each nonzero page of a segment consumes a record or unit of quota. Each page of a segment that is in main memory, whether or not it contains zeros, consumes a record of quota. The consumption of quota is controlled by the existence of quota accounts, possessed by certain directories in the storage system. Every segment in the storage system is said to be charged to some quota account. A quota account consists of two numbers, a quota limit (or "quota" proper) and a "used" (or "records used"), being the sum of all of the quota consumptions of all segments charged to this account.

As page control is responsible for the creation and destruction of pages, page control bears the ultimate responsibility for quota management. When page control creates or destroys a page, not only must the "records used" of the concerned segment be adjusted, but the quota account of the directory against which the segment's records are charged appropriately adjusted. Since page creation happens in the page-reading primitive, and destruction on the page-writing and truncation functions, any quota cell against which any active segment's records are charged must be in wired storage, so that it may be referenced via these functions, which run as part of page control, with the page table lock locked. Each AST entry has room for a quota cell, and thus, only the quota cells in ASTEs of directories bearing quota accounts are actually used (although the "records used" field of each directory which does not have a quota account is maintained as though it did; this allows the "get_quota" (gq) command to be used on such directories to report page record usage). The need to keep these quota cells in wired storage requires that all superior directories of a given segment be active. This is the current reason for this need. For each page creation or destruction, page control chases the chain of ASTEs of superior directories of a segment until an ASTE with a quota account (aste.tqsw on) is found; by definition, this is the quota account to which the segment's records are charged.

There are two classes of record quota, segment quota and directory quota, being for pages of non-directory and directory segments, respectively. Each quota cell in the system (in ASTEs and VTOCEs) has space for both types of quota accounts. A directory may have either or both or neither type of quota account. Page control charges segments' pages against the correct type of quota account, as appropriate. However, when creating a page of a directory, quota checking (i.e., checking the appropriate account to see if the quota limit has been passed) is suppressed (as are all page faults with an effective reference ring of zero). This means that directory quota limiting is essentially not implemented in release 5.0; this is due to the impropriety of signalling record_quota_overflow as a means of conveying the exceeding of such limits to directory control.

Since the checking and adjustment of quota cells by page control is performed under the page table lock, adjustment of quota cells via user command or other storage system action must be protected by the page table lock (although some higher lock could have been devised, one would still have to be wired and masked to lock this lock). Thus, page control provides a procedure, "quotaw" (the "w" is for wired), in bound_page_control, which locks the page table lock and adjusts quota cells. It is given as a parameter the AST entry pointer for either a directory whose quota cells are being adjusted, or in some cases the AST entry pointer for a segment, the quota account against which it is charged having to be located and adjusted. This means that all quota cell adjustment must be performed on active directories only; directory control/segment control ensure that this is the case by activating directories to be involved in quota transactions, and passing pointers to their ASTEs to quotaw. The utility program "quota" in bound_file_system is the user visible interface to quota cell manipulation; it identifies directories given their pathnames, locks them and checks access to manipulate quota, handles "master directory quota," activates directories to be involved in quota transactions (using the "activate" primitive; see "Significance of 'activate'", Section IV), and finally, with the AST locked, passes ASTE pointers to quotaw. Segment control, in the segment truncation primitive, similarly activates the parent directory of a segment being truncated, in order to pass its ASTE pointer to quotaw to adjust the relevant quota cell.

The program quotaw has three general entries, "cu", "sq", and "mq", to change the records-used of a quota account, set the quota limit of a quota account, and "move quota" between a quota account and an inferior quota account (decrease limit of one by a certain amount, incrementing inferior's limit by that much). In all cases, a number of records, a quota type (directory or segment quota), and a pointer to the ASTE of a segment (which is charged against the relative quota account, or the directory ASTE itself) is provided as input. The "mq" entry (move quota) takes another ASTE pointer in addition, being the "inferior" ASTE to which quota is to be moved. All of the entries lock the page table lock (via pmut$lock_ptl, see Section VIII) and loop up the AST parent threads to find the correct quota account, and perform the necessary adjustment. The "cu" entry, (change-used, which is generally used to adjust the records-used number of an account) also supports the function of checking whether or not a contemplated change in records-used is valid; an input switch specifies this. All of the entries return a standard status code.

The program quotaw also has a "side-door" (quotaw$cu_for_pc) which is used by the segment-truncation function (in pc$truncate) and the deactivation-time service (pc$cleanup) to adjust quota cells when these functions destroy (or find zero) pages. This entry is similar to quotaw$cu, except that it is called with the page table lock locked, and the process wired and masked, and returns with these circumstances prevailing as well.

SECTION XII

RING ZERO VOLUME MANAGEMENT

INTRODUCTION AND OVERVIEW

Volume management concerns itself with the relation between physical volumes and logical volumes, and between physical volumes and disk drives. It is the responsibility of volume management to ensure the integrity of information upon a given physical or logical volume, and to perform the necessary binding and unbinding operations in the supervisor when volumes are mounted and demounted.

Volume management, as described in these sections, does not concern itself with the operator interface for mounting and demounting, nor the completeness or registration of logical volumes.

There are four sections iı this portion of the book:

Section XII	Introduction and Overview
Section XIII	Data Bases of Ring 0 Volume Management
Section XIV	Operations of Ring 0 Volume Management
Section XV	Interaction of the Physical Volume Salvager with the Storage System

Many of the operations that may be considered part of volume management, such as segment moving and physical volume assignment, are covered in Section IV.

Unlike the other subsystems described in this document, no sections describing functions or services of volume management are provided. The only services provided are the mounting and demounting of physical and logical volumes, and the determination of whether or not a given physical or logical volume is in fact mounted. There are no lower-level mechanisms to speak of. Thus, the functions and services of ring zero volume management are placed together under the section "Operations of Ring Zero Volume Management."

CONCEPTS

A physical volume is a disk pack that is described by registration information maintained by the volume registration package in ring 1. A physical volume is divided into records of 1024 words each. These records may contain pages of segments, or be part of the VTOC (Volume table of contents) of the physical volume, be part of partitions, or be part of the volume header. The VTOC consists of entries (VTOCEs), five to a page, that describe the segments whose pages are on this physical volume, one VTOCE per segment. The partitions are conterminous regions of disk set aside for special use, such as FDUMP images and the syserr log. The volume header, which is at a fixed location on the disk, contains information describing the extent and location of the partitions and VTOC, as well as a map (the volume map) of which records are in use by pages of segments. All of the area not in use by the volume header, VTOC, or partitions is called the paging region of the volume, and it is from here that records are used by pages of segments. Every segment described by a VTOCE on this pack has all of its pages on this pack; no segment has pages on several packs. The most important data item in the volume header is the volume label, or label. This data item contains duplication of the registration information kept for this volume, identifying it, and a history of the last use of this volume by Multics. It is this latter information that allows volumes to be used in a consistent fashion across crashes.

The Root Physical Volume, or RPV is the physical volume that contains the root directory, ">", as one of its segments. It is special-cased by the system in many ways. It is the only volume known to the system at the time it is bootloaded. Another segment on the RPV is the disk table, a ring-1 data base that describes the drives on which all packs were located during the last bootload. From this data base, the ring 1 software can bring other volumes into use at the time the system is brought up.

A logical volume is a user-visible collection of physical volumes, designated as such by the volume registration data in ring 1. With the exception of the RPV, no physical volume may be in use by Multics unless all other physical volumes in the logical volume to which it belongs are also in use. Thus, logical volumes are mounted and demounted as a unit. Each directory in the storage system hierarchy has a unique logical volume on whose physical volume all segments immediately contained in that directory reside. This logical volume is called the son's logical volume of that directory.

The root logical volume, or RLV, is that logical volume of which the RPV is a member. The RLV is the only logical volume that may be partially in use. The RLV is the only logical volume that contains directory segments. Although the segments inferior to any given directory reside on the son's logical volume of that directory, the directories reside on the root logical volume. Operationally, the root logical volume is the only one necessary to bring the system up to ring 4 command level, through answering-service startup. The root physical volume is the only physical volume necessary to bring-the system up to ring-1 command level.

To mount a physical volume is to physically place it on a drive and cycle up that drive. This action is performed by the operator, not by software.

To accept a physical volume is to make the necessary calls to the supervisor, for a drive on which a given physical volume has been mounted, to establish in the supervisor the binding between that drive and the physical volume on it. Critical in this binding is the placement of the 36-bit Physical Volume ID (PVID) read from the label of that physical volume in a table entry (the PVTE) associated with that drive. The descriptions of segments in directory branches are in terms of these physical volume IDs, and VTOC indices. Thus, placing this ID in this table entry indicates that the volume is indeed online, and segments on it may be used (via the process of activation, see Section II.

A logical volume is mounted (or "mounted to the system") when all of the physical volumes in it are mounted and accepted, and calls have been made to the supervisor to establish the presence of this complete logical volume on line. Unless a logical volume is mounted, the system will refuse to honor initiations of segments, segment control calls, and segment faults for segments on physical volumes of that logical volume, even though the physical volumes may have been accepted. It is via this policy that the usage of "incomplete" logical volumes is interdicted. One exception to this rule is the root logical volume.

It is mounted even if it is incomplete; it is mounted as soon as the RPV is accepted in system initialization. The system maintains a table of mounted logical volumes, the LVT, or Logical Volume Table. Each entry in it, or LVTE, describes one mounted logical volume, containing per-logical volume information, as well as the start of a chain of PVTEs of accepted physical volumes in this logical volume. Ring 1 will not make the call to mount a logical volume until it has verified that all physical volumes known (from the volume registration data) in it have been accepted.

The system maintains a table, the Physical Volume Table or PVT, containing information about each accepted physical volume. It has one entry, or PVTE, per each disk drive known to the system. This entry contains information about the physical volume mounted on that drive, including its PVID and Logical Volume ID (LVID) of the logical volume to which it belongs. Parameters about this volume, read in from its volume header at the time it was accepted, that are used by page control and segment control in dealing with segments upon this volume and their pages, are kept in the PVTE. The PVTE also contains information used by page control and the disk DIM describing the physical drive associated with the PVTE, such as its device number and device type.

A logical volume may be mounted to a given process or not, if it is mounted at all. A mounted logical volume is mounted to a given process if it is either a public logical volume, or (a private logical volume) a call has been made by RCP in ring 1 in that process to attach the private logical volume to the process. RCP will allow a private logical volume to be attached to a process pendant on whether or not that process has access to the logical volume, as defined by the ACS (access control segment) for that volume, created by the ring 1 registration software. Unless a logical volume is mounted to a given process, the process will act as though the logical volume were not mounted at all; segment faults and initiations are not honored, and segment control calls may not be made. Thus, only those processes selected by the ACS of a private logical volume may use the segments on it, while all processes may use the segments on a public logical volume, subject to the normal Multics access control mechanism and AIM access control. The table of private logical volumes attached (and therefore mounted) to a process is kept in a region of the KST (Known Segment Table) of a process. This set of logical volumes attached to the process is necessarily a subset of the logical volumes that are mounted (to the system), as kept in the LVT. A logical volume that was attached by a given process may be detached by that process, via a call through RCP in ring 1. When this occurs, the logical volume is no longer mounted to the process and segments on it may no longer be used by this process (a local setfaults operation (see Section II) is performed).

A logical volume may be demounted by calling the supervisor to remove the Logical Volume Table entry for it. This prevents further attachments to the logical volume, but does not stop use of the segments in it until each physical volume in the logical volume is demounted. These calls are made by ring 1 volume management in the initializer process.

A physical volume is demounted by making a call to the supervisor (from ring 1 of the initializer process) to stop all processes from using segments on this volume, deactivate all of these segments, flush VTOC buffers of all information relating to this volume, update the volume header of the volume, and remove information from the PVTE for the drive containing that volume which describes it. This unbinds the volume from the drive. At system shutdown time, all volumes are demounted, the RPV being demounted last. At this time, however, a modified form of deactivation is performed that does not involve freeing AST entries or dealing with AST threads (see Section IV).

PREACCEPTANCE

The RPV is accepted, like all other volumes, before segments on it are available for use. For the RPV, this happens during collection 2 of system initialization. However, the RPV is used prior to this, but not segments on it. All of this activity occurs in the hardcore partition of the RPV, and consists of the running of initialization from the running of the program init_pvt up until the acceptance of the RPV. This may involve a volume salvage of the RPV if it had not been shutdown properly during the last bootload. The hardcore partition exists to satisfy the need for a fixed, usable area, for paging by the supervisor, when the validity of the RPV volume map may not be trusted. (See Section VII for more detail on this.)

The point in collection 1 initialization at which the use of the hardcore partition is established, and thus the first paging in initialization begins, is called the preacceptance of the RPV. The RPV label is read, the partition extents on it determined, and the use of the hardcore partition set up. Global system parameters in the FSDCT, relevant to the success of the last shutdown, are determined from the RPV label, as well as the active/unflushed status of any paging device that must exist.

Between the preacceptance and acceptance of the RPV, no VTOC I/O nor segment faults occur on the RPV. No activations occur, nor is the paging region nor VTOC used at all (except the former by the physical volume salvager). The bit-map for the RPV during this time is not the bit-map from the volume-map of the RPV, but rather a special one fabricated by the preacceptance code. It defines the hardcore partition.

The preacceptance of the RPV is performed in the program init_pvt.

SECTION XIII

DATA BASES OF RING ZERO VOLUME MANAGEMENT

This section describes the detailed structure and functic supervisor data bases that are used to manage the set of physical volumes known to the supervisor. A large part of the visible i volume management, however, is that presented by the ring 1 volum package, responsible for the operator interface and volume functions. The data bases of these functions, in particular the Dis Logical Volume Attach Table, and the Registration Files, are n herein.

Some of the critical data bases used by ring zero volume manage seen in the Multics supervisor at all; they are resident on regions and are explicitly read in and written out at the times that inspected or modified. These data bases, in the volume header of reside at fixed record addresses on each pack, given in the i disk_pack.incl.pl1. These data bases will be described first.

VOLUME LABEL

The volume label resides on the first Multics record of each st physical volume. It is generated by the program init_disk_pack volumes except the RPV, in which case it is generated by init_empty_root) in the ring 1 volume management environment. It is the time a volume is accepted, and written out at the time it is de is also written out at the time it is accepted, to indicate that the not shut down, until it is written out at the time it is demounted record is divided into five regions, on sector boundaries:

1. GCOS region, sectors 0 to 4 (label.gcos). This region is entirely, as the Series 6000 GCOS system uses this part of label area. Avoiding use of this region avoids accidental of Multics data by labeling a pack under GCOS at a site operating systems, and allows some future compatability.

2. Permanent region, sector 5 (label.Multics to label.pad1). is per-physical volume information that is never char written out identically from the copy read in every time label is written. Were it possible to write-protect sing this sector would be so protected at the time the pack was for Multics use. This is permanent identifying information some of it is subject to change by the disk rebuilder).

3. Dynamic information, Sector 6 (label.time_mounted to l This information relates to the use of this physical volum last mounted, demounted, etc. This information allows t system to ensure integrity of data on the physical volu dynamic state.

4. Root information, Sector 7 (label.root to label.pad3). This information is defined only on the root physical volume (RPV) of a hierarchy. It is dynamic information about the entire storage system hierarchy: how successfully if at all it was shut down, and information relative to crash recovery, and bootstrapping the initialization of the directory hierarchy at bootload time.

5. Partition map (sector 10 (octal) (label.parts). A map giving the location and length of any partitions defined on this physical volume. This information is set up at the time that a volume is initialized, and never changed (except by the disk rebuilder).

   The rest of the label record (sectors 11-15, octal) is reserved for future expansion.

Detailed breakdown of the label:

```
dcl 1 label based (labelp) aligned,


    2 gcos (5*64) fixed bin,


    2 Multics char (32) init ("Multics Storage System Volume"),
    2 version fixed bin,
    2 mfg_serial char (32),
    2 pv_name char (32),
    2 lv_name char (32),
    2 pvid bit (36),
    2 lvid bit (36),
    2 root_pvid bit (36),
    2 time_registered fixed bin (71),
    2 n_pv_in_lv fixed bin,
    2 vol_size fixed bin,
    2 vtoc_size fixed bin,
    2 not_used bit (1) unal,
    2 private bit (1) unal,
    2 flagpad bit (34) unal,
    2 max_access_class bit (72),
    2 min_access_class bit (72),
    2 password bit (72),
    2 pad1 (16) fixed bin,
    2 time_mounted fixed bin (71),
    2 time_map_updated fixed bin (71),
    2 time_unmounted fixed bin (71),
    2 time_salvaged fixed bin (71),
    2 time_of_boot fixed bin (71),
    2 pd_time fixed bin (71),
    2 last_pvtx fixed bin,
    2 pad1a fixed bin,
    2 n_bad_tracks fixed bin,
    2 err_hist_size fixed bin,
    2 time_last_dmp(3) fixed bin(71),
    2 dmpr_hd(2) fixed bin,
    2 bk_dmpr_hd(2) fixed bin,
    2 curn_dmpr_item(3) fixed bin,
    2 pad2 (35) fixed bin,
    2 root,
      3 here bit (1),
      3 root_vtocx fixed bin (35),
      3 shutdown_state fixed bin,
      3 pd_active bit (1) aligned,
      3 disk_table_vtocx fixed bin,
      3 disk_table_uid bit (36) aligned,
      3 esd_state fixed bin,
    2 pad3 (60) fixed bin,
    2 nparts fixed bin,
    2 parts (47),
      3 part char (4),
      3 frec fixed bin,
      3 nrec fixed bin,
      3 pad5 fixed bin,
    2 pad4 (5*64) fixed bin;
```

label.gcos
Reserved for compatability with the GCOS system. See above.

label.Multics
Contains the character string "Multics Storage System Volume" on every pack. Used for gullibility checks against unlabeled packs, and by resource control to avoid accidental overwriting or disclosure of information on storage system packs.

label.version
Currently must be 1.

label.mfg_serial
Intended to be the manufacturer's serial number for a pack, this is currently set to be physical volume name.

label.pv_name
The physical volume name of the pack.

label.lv_name
Is the logical volume name of the logical volume to which this physical volume belongs.

label.pvid
Is the 36-bit unique ID (PVID) of the physical volume. This same number is contained in the directory branches of all segments contained on this physical volume.

label.lvid
is the 36-bit unique ID (LVID) of the logical volume to which this physical volume belongs. It is contained in all directories for which that logical volume is the sons-logical-volume.

label.root_pvid
is the 36-bit PVID of the RPV of the hierarchy of which this volume is part. This information defines which packs belong to a given hierarchy.

label.time_registered
is currently set to the 52-bit clock time that the volume was initialized for use by the storage system.

label.n_pv_in_lv
is currently not used.

label.vol_size
is the number of Multics records physically available on this volume, regardless of how they are used.

label.vtoc_size
is the number of Multics records used by the Volume Table of Contents (VTOC) <u>and</u> the volume header.

label.private
is "1"b if and only if the logical volume to which this physical volume belongs is a private logical volume.

label.max_access_class
is the maximum AIM access class for segments on the logical volume to which this physical volume belongs. No segments of higher access class (in the AIM sense) can be allocated on that logical volume.

label.min_access_class
is the minimum AIM access class for that logical volume.

label.password
is currently not used.

label.time_mounted
is the last time that this physical volume was accepted by the supervisor.

label.time_map_updated
is the last time at which the label of this physical volume was written. Please note that this name is very misleading; this extremely important quantity, which determines the need to volume salvage (see Section XIV) should be thought of as "label.time_label_written".

label.time_unmounted
is the last time that this volume was demounted, including for shutdown.

label.time_salvaged
is the time that the label was last written out at the completion of processing of this volume by the physical volume salvager.

label.time_of_boot
is the time recorded as "time of bootload" for the system (in the FSDCT) for the last Multics bootload that accepted this volume.

label.pd_time
is the time ("paging device time") identifying the last instance of the paging device to which this physical volume was exposed. By comparing this paging device time to that of an unflushed paging device, repatriation of records may be accomplished. (See "Post-Crash PD Flush" in Section IX).

label.last_pvtx
is the physical volume table index (PVTX) of the drive on which this volume resided at the last time it was accepted by the storage system. By comparing this value with that in paging-device map entries in the PDMAP of the instance of the paging device identified by label.pd_time, repatriation of records may be accomplished. (See "Post-Crash PD Flush" in Section IX.)

label.n_bad_tracks
not currently used.

label.err_hist_size
not currently used.

label.time_last_dmp
reserved for the physical volume dumper.

label.dmpr_hd
reserved for the physical volume dumper.

label.bk_dmpr_hd
reserved for the physical volume dumper.

label.curn_dmpr_item
reserved for the physical volume dumper

label.root
substructure covering the "root information" in the label.

label.here
identifies this physical volume as the RPV of a hierarchy (although other tests will suffice).

label.root_vtocx
is the index in the VTOC of this pack of the directory ">".

label.shutdown_state
is set to various values during the course of shutdown. It is essentially obsolete.

label.pd_active
is "1"b if the system has an active or unflushed paging device. If, at bootload time, when the RPV is interrogated, this bit is on, the system has an unflushed paging device.

label.disk_table_vtocx
is the index in the VTOC of this pack of the segment ">disk_table". Reserved for future use.

label.disk_table_uid
is the unique segment ID of the segment ">disk_table". Reserved for future use.

label.esd_state
is set to zero by the stages of normal shutdown, and to nonzero values by the stages of emergency shutdown. The nonzero value of this variable at the time the RPV is first inspected during a bootload implies that the previous bootload had a successful emergency shutdown. This triggers RPV salvage to collect pages of RPV parasite segments. (See Section VII.)

label.nparts
is the number of partitions on this volume.

label.parts
is an array defining the partitions on this volume. The number of valid entries is given by label.nparts.

label.parts.part
is the four-character ASCII name of a partition.

label.parts.frec
is the first record number on this pack of the partition.

label.parts.nrec
is the number of Multics records used by this partition.

VOLUME MAP

The volume map details which records of the paging region of a physical volume are in use. Although this information may be derived from analysis of every VTOCE on the pack, it is duplicated in the volume map so that record allocations can be performed by page control without inspection of every VTOCE on the pack. If a pack is not shut down properly, this information is considered to be wholly invalid, and is reconstructed by the physical volume salvager via inspection of every VTOCE on the pack. When a volume is accepted, the information in the volume map is copied into the free-store bit-map (see "Disk Record Allocation" in Section VIII) for the drive on which the pack is mounted. It is written back to the volume map on the disk at the time the volume is successfully demounted. The information in the header of the volume map is copied to and from the so-called "fsmap parameters" (see Section VI) in the PVTE for that drive.

```
dcl 1 vol_map based (vol_mapp) aligned,

    2 n_rec fixed bin(17),
    2 base_add fixed bin(17),
    2 n_free_rec fixed bin(17),
    2 bit_map_n_words fixed bin(17),
    2 pad (60) bit(36),
    2 bit_map (3*1024 - 64) bit(36) ;
```

vol_map.n_rec
: is the number of records in the paging region of the pack, and hence, the number of records represented by the volume map.

vol_map.base_add
: is the Multics record number of the first record of the paging region of the pack, and thus the record number of the first bit in the volume map.

vol_map.n_free_rec
: is the number of records in the paging region of the pack which are not allocated. It should be equal to the number of bits which are "1"b in the volume map.

vol_map.bit_map_n_words
: is the number of words in the volume map's bit map. If the number of records in the paging region is not a multiple of 32, the last bits of the last word (the "fsmap tail") will be "0"b, but are not considered part of the bit map.

vol_map.bit_map
: is the array of words that constitute the bit map described by the parameters just described. Neither the first bit nor the last three bits of each word are used, being "0"b in all cases. This leaves 32 bits per word, representing 32 Multics records in each word of the bit map. The value "1"b indicates a free record, and "0"b indicates a record in use.

The volume map is considered to be wholly invalid between the time that a physical volume is accepted and the time that it is successfully shut down or salvaged (see Section XIV).

VTOC HEADER

The VTOC header describes the extent, and global parameters, of the VTOC of a pack. These parameters are copied into the PVTE for the drive on which the pack is mounted at the time it is accepted, and update to the VTOC header from there every time the label is written out. Like the volume header, it is considered wholly invalid (at least the dynamic parameters therein) from the time the volume is accepted to the time the volume is successfully demounted, and must be reconstructed by the physical volume salvager if the volume is accepted without having been shut down.

vtoc_header.version
: currently must be 1.

vtoc_header.n_vtoce
: is the number of VTOC entries (VTOCEs) in the VTOC of this pack, used or free. This is constant, modulo the disk rebuilder.

vtoc_header.vtoc_last_recno
is the Multics record number of the last record occupied by the VTOC of this pack. The first record is currently a constant VTOC_ORIGIN, defined in disk_pack.incl.pl1.

vtoc_header.first_free_vtocx
is the VTOC index of the VTOCE on this pack which is the first in the free chain. This index is maintained in the PVT by the VTOC manager while the pack is in use.

The rest of the VTOC header record is reserved for the physical volume dumper.

BAD TRACK LIST

This information is not currently maintained by Multics.

FSDCT

The letters "FSDCT" stand for "file system device configuration table." This name is largely historical, for the segment that contains free-store bit maps and per-hierarchy information, and has ceased to have any significance.

The FSDCT contains two distinct regions. The FSDCT header contains global data about the state of ring zero volume management. Much of it is derived from the RPV label at the time the RPV is preaccepted during collection 1. Much of it is derived from CONFIG cards, and much of it is written out to the RPV label at various stages of shutdown. It defines the state of shutdown, and the state of ring zero with respect to volume management.

The region of the FSDCT beyond the header consists of the bit maps for disk record allocation for each drive. One region is allocated for each drive, and the volume map bit map from each physical volume is copied in at the time that the volume is accepted. The relative offset of the bit map for each region is defined by the field pvte.fsmap_rel in the fsmap parameters in the PVTE for that drive. The FSDCT is a pageable data base; the withdrawal of disk records from it at page fault time is accomplished via an esoteric maneuver described fully in Section VIII.

The following include file and discussion describe the FSDCT header.

fsdct.shutdown_state
is zero while Multics is running, and set to various nonzero values during normal and emergency shutdown. It is updated to the field label.root.shutdown_state each time the label of the RPV is written out.

fsdct.oos_dir
is obsolete.

fsdct.esd_state
is zero while Multics is running, and set to various nonzero values during emergency shutdown. It is updated to the field label.root.esd_state in the label of the RPV each time the label of the RPV is written out.

fsdct.prev_shutdown_state
is the value of the label.root.shutdown_state in the label of the RPV at the time that the RPV is preaccepted during collection 1 initialization. Thus, it describes the shutdown state of the previous bootload of this hierarchy.

fsdct.prev_esd_state
is the value of label.root.esd_state in the label of the RPV at the time that the RPV is preaccepted during collection 1 initialization. Thus, it tells whether or not this hierarchy last witnessed a successful emergency shutdown.

fsdct.rpvs_requested
is set to "1"b if the operator issued a BOOT RPVS request to boot the system, requesting an RPV salvage (RPVS).

fsdct.root_lvid
is the 36-bit Logical Volume ID (LVID) of the RLV of this hierarchy.

fsdct.root_pvid
is the 36-bit Physical Volume ID (PVID) of the RPV of this hierarchy.

fsdct.root_pvtx
is the physical volume table index (PVTX) of the drive on which the root physical volume (RPV) is mounted. This value is duplicated for various functions in the SST, as sst.root_pvtx. It is derived from the ROOT CONFIG card.

fsdct.root_vtocx
is the index in the VTOC of the RPV of the directory ">", copied from label.root.root_vtocx on the RPV label.

fsdct.rlv_needs_salv
is "1"b if a volume of the RLV needed a salvage at the time it was accepted, and was salvaged. This bit informs the operator interface that a hierarchy salvage of selected directories must be performed at system startup time. (This is because all directories reside on the RLV, and the fact that some volumes of it were not properly shut down may indicate that some directories were damaged.)

fsdct.n_volumes
is not used.

fsdct.dump_part_pvtx
is the physical volume table index (PVTX) of the drive on which the volume with the system's DUMP (BOS FDUMP) partition exists. This drive is selected by the PART DUMP card in the CONFIG deck. It is zero if there is no DUMP partition.

fsdct.dump_part_frec
is the first record number of the DUMP partition, if one exists, on the pack on the drive selected by fsdct.dump_part_pvtx.

fsdct.syserr_log_pvtx
is the physical volume table index (PVTX) of the drive on which the volume with the system's syserr log partition exists. This drive is selected by the PART LOG card. It is zero if the system is not using syserr logging.

fsdct.syserr_log_frec
is the first record number of the syserr log partition, if one exists, on the pack on the drive selected by fsdct.syserr_log_pvtx.

fsdct.syserr_log_nrec
is the number of records in the syserr log partition, if one exists, otherwise zero.

fsdct.free

is not used.

fsdct.hc_exists

is obsolete, and is always "1"b.

fsdct.hc_using

is set on during the preacceptance of the RPV in collection 1, and turned off during the acceptance of the RPV in collection 2. It indicates that the system is running totally in the hardcore partition.

fsdct.hcp_frec

is the first record number on the RPV of the hardcore partition. The RPV must have a hardcore partition defined on it. This number is obtained from the RPV label during preacceptance of the RPV.

fsdct.disk_table_vtocx

is the index in the VTOC of the RPV of the VTOCE describing the segment ">disk_table". It is read in from the root area of the RPV label, but is not now used.

fsdct.disk_table_uid

is the unique segment ID of the segment ">disk_table". Not now used.

fsdct.pd_active

is "1"b if and only if the system has an active paging device. It is set during RPV preacceptance in collection 1, and by shutdown, and is managed dynamically by the cross-bootload paging device management policies. (See Section IX for a discussion of active and unflushed paging devices.)

fsdct.rpv_needs_salv

is set to "1"b during RPV preacceptance if the RPV was not properly shut down during the last bootload. This triggers an RPV salvage later.

fsdct.pd_unflushed

is set to "1"b if and only if the system has an unflushed paging device (see Section IX.)

fsdct.pd_time

is the paging device time identifying the instance of the paging device to which this hierarchy was last exposed. If this bootload never had an unflushed paging device, this is the same as fsdct.time_of_bootload. If the paging device is unflushed, this variable has the value of the variable fsdct.time_of_bootload from the bootload during which that instance of the paging device was active. Otherwise, if the paging device was dynamically enabled during this bootload, this is the time at which that was done. (See Section IX, "Post-Crash PD Flush.")

fsdct.old_root_pvtx

is the value of the cell label.last_pvtx in the label of the RPV. It is used to repatriate RPV pages during acceptance of the RPV. (See Section IX, "Post-Crash PD Flush.")

fsdct.maps

is (*is*, not *contains*) the first word of the bit-map region of the FSDCT.

PHYSICAL VOLUME TABLE (PVT)

The physical volume table, or PVT, is the single most important data base of ring-zero volume management. It contains an entry, or PVTE, for each disk drive known to the system (including so-called "I/O drives"). It also has an entry for the bulk store subsystem (at the end) if one exists, as this is required by page control. The information in the PVTE for each drive describes information needed by the disk DIM to describe that drive with respect to the former's data bases. This includes the device number and subsystem name, as well as the device type. This information stays constant in each PVTE. The PVTE, however, also is filled in with information about the volume mounted on the corresponding drive at the time that such volume is accepted. This information consists of the quantities from the volume's volume header, specifically the VTOC header and volume map. This data is used by segment control and page control to manage the VTOC and the free store bit-map of the volume. Included in the PVTE is also data that describes a region of the FSDCT which is used as the bit-map for each volume mounted on that drive. This information is permanent. The specific parameters for whatever bit-map may be there as a given volume is used is not permanent. The PVT is a paged, wired, deciduous segment, which is used by page control, and thus must not be pageable.

```
dcl 1 pvt based (pvtp) aligned,


    2 n_entries fixed bin (17),
    2 max_n_entries fixed bin (17),
    2 n_in_use fixed bin (17),
    2 rwun_pvtx fixed bin,
    2 pad (4) bit (36),


    2 array (0 refer (pvt.n_entries)) like pvte;


dcl 1 pvte based (pvtep) aligned,


    2 pvid bit (36),


    2 lvid bit (36),


    2 dmpr_in_use (3) bit (1) unaligned,
    2 pad3 bit (24) unaligned,
    2 brother_pvtx fixed bin (8) unaligned,


    2 devname char (4),


    (2 device_type fixed bin (8),
    2 logical_area_number fixed bin (8),
    2 used bit (1),
    2 storage_system bit (1),
    2 permanent bit (1),
    2 testing bit (1),
    2 being_mounted bit (1),
    2 being_demounted bit (1),
    2 check_read_incomplete bit (1),
    2 device_inoperative bit (1),
    2 rpv bit (1),
    2 paging_device bit (1),
    2 salv_required bit (1),
    2 being_demounted2 bit (1),
    2 vol_trouble bit (1),
    2 vacating bit (1),
    2 pad bit (4),


    2 first_free_vtocx fixed bin (17),
    2 n_free_vtoce fixed bin (17),


    2 vtoc_size fixed bin (17),
    2 vtoc_segno fixed bin (17),


    2 fsmap_rel bit (18),
    2 bad_addrs_consecutive fixed bin (17),
    2 dbmrp (2) bit (18)) unaligned,


    2 curwd bit (18),
    2 wdinc bit (18),
    2 temp fixed bin,
    2 baseadd fixed bin,
    2 tablen bit (18) unaligned,
    2 tablen_allocation fixed bin (17) unaligned,
    2 nleft fixed bin,
    2 relct fixed bin,
    2 totrec fixed bin,


    2 dim_info bit (36),


    2 curn_dmpr_vtocx (3) fixed bin unaligned,
    2 n_vtoce fixed bin unaligned;
```

pvt.n_entries
is the number of entries, used or otherwise, in the PVT array.

pvt.max_n_entries
is the same as pvt.n_entries.

pvt.n_in_use
is number of entries corresponding to accepted volumes.

pvt.rwun_pvtx
is the PVT index of a drive (only one may be in this state at a time) expecting an interrupt from the I/O interfacer for cycling down the drive at demount time.

pvt.array
is the array of PVTEs.

pvte.pvid
is the 36-bit Physical Volume ID (PVID) of the accepted volume mounted on this drive, zero if none.

pvte.lvid
is the 36-bit logical volume ID (LVID) of the logical volume to which the accepted volume on this drive belongs, zero if none. As pvte.pvid, this parameter is read in from the volume label at acceptance time.

pvte.dmpr_in_use
is reserved for the physical volume dumper.

pvte.brother_pvtx

is the PVT index of the next volume in the chain of physical volumes belonging to the same logical volume as the one to which the accepted volume on this drive belongs.

pvte.devname

is the four-character ASCII name of the disk subsystem to which this drive belongs.

pvte.device_type

is the hardware device type, as defined in fs_dev_types.incl.pl1, of this disk drive.

pvte.logical_area_number

is the hardware drive number of this disk drive.

pvte.used

is "1"b if and only if there is an accepted volume on this drive. It is off in the PVTE of the RPV until the RPV has been accepted.

pvte.storage_system

is "1"b for a drive that is not an "I/O drive" defined by a "UDSK" CONFIG card.

pvte.permanent

is "1"b for a drive designated by a PART card, and is also "1"b for the RPV. No pack except the one mounted there at bootload time may ever be mounted on this drive during this bootload.

pvte.testing

is set to "1"b by the program read_disk (see Section XIV) before a special call is made to disk_control. This bit tells the disk_dim interrupt side to set pvte.device_inoperative according to the relative success of a "request status" operation on this drive. Disk control turns off this bit when the latter bit has been set.

pvte.being_mounted

is "1"b during the acceptance of a volume on this drive. Primarily informational.

pvte.being_demounted

is set to "1"b at the start of the demount procedure for a volume on this drive. Prevents activations of segments on this volume. (See Sections IV and XIV.)

pvte.being_demounted2

is set to "1"b during the latter part of the demount procedure for a volume on this drive. Prevents VTOC I/O from being initiated. (See Sections IV and XIV.)

pvte.check_read_incomplete

causes page control to store special patterns into core frames into which records of this volume will be read, and check for their presence at the posting of the operation. There is no way to turn this feature on other than patching this bit.

pvte.device_inoperative

is used by the program read_disk, along with the bit pvte.testing, to determine if a drive is operative. (See pvte.testing, above, and Section XIV.)

pvte.rpv

is "1"b in the PVTE of the RPV.

pvte.paging_device

is "1"b in the PVTE of the bulk store subsystem, required by page control to perform abs-seg I/O on the PDMAP.

pvte.salv_required
is set to "1"b during the acceptance of a volume if it was not properly shutdown during its previous use, and thus required and received a volume salvage.

pvte.vol_trouble
is set by various recovery procedures (and ESD) if there is reason to believe that an operation upon the VTOC of a volume is interrupted in such a way that the volume is inconsistent, and will require a volume salvage at some time. This bit being on causes the volume to be shut down in such a way (at the time it is demounted) that it will appear that it was not properly shut down, the next time it is accepted, and thus require and receive a volume salvage.

pvte.vacating
inhibits VTOC allocation (segment creation) upon this physical volume. It is used by the on-line physical volume utility, sweep_pv (see the Multics Operators' Handbook, Order No. AM81, and Section IV, "Segment Control Services" for sweep_pv).

pvte.first_free_vtoc
is the index, in the VTOC of the physical volume accepted on this drive, of the VTOCE that is the head of the free VTOCE chain for this volume. It is maintained by the VTOC manager (see Section III), and copied to and from vtoc_header.first_free_vtocx and acceptance and demount time, respectively.

pvte.n_free_vtoce
is the number of free VTOCEs in the VTOC of the physical volume accepted on this drive. It is maintained by the VTOC manager, and copied to and from vtoc_header.n_free_vtoce at acceptance and demount time, respectively.

pvte.vtoc_size
is the number of Multics records in the VTOC and volume header of the physical volume accepted on this drive. Read in at acceptance time from label.vtoc_size.

pvte.vtoc_segno
is a temporary used by the physical volume salvager.

pvte.fsmap_rel
an "fsmap parameter," is the relative offset into the FSDCT of the region allocated for bit-maps for volumes on this drive.

pvte.bad_addrs_consecutive
is not used.

pvte.dbmrp
is reserved for the physical volume dumper.

pvte.tablen_allocation
an "fsmap parameter," is the length, in words, of the region in the FSDCT allocated for bit-maps for volumes on this drive.

pvte.curwd
pvte.wdinc
pvte.temp
pvte.baseadd
pvte.tablen
pvte.nleft
pvte.relct
are the "fsmap parameters," copies of information in the volume map of the physical volume mounted here, and information needed by and maintained by the free-store allocation algorithm. These fields are described in the PVTE writeup in Section VI.

pvte.dim_info

is information stored by disk DIM initialization for this drive, which the disk DIM needs to perform address computations on this drive, and identify its subsystem.

pvte.curn_dumper_vtocx

is reserved for the physical volume dumper.

pvte.n_vtoce

is the number of VTOCEs, free or used, in the VTOC of the physical volume accepted on this drive.

LOGICAL VOLUME TABLE (LVT)

The logical volume table (LVT) is used to describe all mounted logical volumes. It contains all per-logical-volume data for such logical volumes, and contains threads of the PVTEs of accepted physical volumes that are members of each logical volume. The logical volume ID, however, is duplicated in each PVTE for physical volumes in that logical volume. This enables the segment creation function to operate without a lock. (See Section IV for a description of this activity.) The LVT is a pageable segment, used at segment creation and segment moving time, as well as the time that logical volumes are mounted and demounted (see Section XII).

The LVT contains an entry, a LVTE, for each mounted logical volume. The LVTE for the RLV is set up during initialization (collection 2). The LVTEs for other volumes are set up at the time that they are mounted. The LVT also contains a hash table, hashing LVTEs by their LVIDs of the logical volumes that they describe.

```
dcl 1 lvt aligned based (lvtp),
    2 max_lvtex fixed bin (17),
    2 high_water_lvtex fixed bin (17),
    2 free_lvtep ptr,
    2 pad1 (4) bit (36),
    2 ht (0:63) ptr unal,
    2 lvtes (1:1 refer (lvt.max_lvtex)) like lvte;
```

```
dcl 1 lvte aligned based (lvtep),
    2 lvtep ptr unaligned,
    2 pvtex fixed bin (17),
    2 lvid bit (36),
    2 access_class aligned,
      3 min bit (72),
      3 max bit (72),
    2 flags unaligned,
      3 public bit (1),
      3 read_only bit (1),
      3 pad bit (16),
      3 cycle_pvtx fixed bin (17);
```

lvt.max_lvtex

is the index of the highest-indexed LVTE that can ever exist in this LVT, as defined by the size of the LVT segment.

lvt.high_water_lvtex

is the highest LVT index that was ever used in this bootload. This is a meter.

lvt.free_lvtep
is a pointer to the first in a list of free LVTEs. As they are created as needed, this list is non-empty only if LVTEs have been freed.

lvte.ht
is a hash table, containing pointers to the first LVTEs in the hash threads of each hash equivalence class.

lvt.lvtes
is the array of LVTEs.

lvte.lvtep
for an LVTE in use, is the pointer to the next LVTE in the same LVID hash equivalence class as this one, null if this is the last one. For a free LVTE, it is a pointer to the next LVTE in the chain of free LVTEs, null if this is the last one.

lvte.pvtex
is the PVT index of the first PVTE in the chain of PVTEs for drives containing physical volumes belonging to this logical volume. This chain is threaded through the PVTEs as pvte.brother_pvtx. Zero marks the end of the chain.

lvte.cycle_pvtx
is used by the segment creation function of segment control (see Section IV) to allocate VTOCEs in the logical volume. See that description for its use.

lvte.lvid
is the logical volume ID (LVID) of this logical volume.

lvte.access_class
describes the AIM access class limits of the logical volume.

lvte.public
is "1"b for a public logical volume, "0"b for a private one.

lvte.read_only
is reserved.

PVT HOLD TABLE

The PVT hold table resides in the static section of the program get_pvtx. It is a table of process IDs of processes that start operations on a given physical volume that requires more than one call to the VTOC manager, or a call to the VTOC manager and an action upon the bit-map of the volume. The table consists of an array of marks made by such processes, each mark consisting of the catenation of part of the process' process ID and the PVT index of the volume being modified. These marks are removed when the inconsistent operation is finished.

The purpose of this table is to prevent the volume from being demounted while such an operation is in progress. No process may make a mark in this table if a demount operation has started for a volume on which an operation was about to begin (pvte.being_demounted prevents this). Similarly, the demounting procedure demount_pv will wait for all marks in this table relative to a particular physical volume to vanish before the demount procedure can continue.

If a process suffers a crawlout at such a time that it had made a mark in this table, and thus left a volume in an inconsistent state, not only is its mark or marks removed from the table, but that volume is scheduled for a salvage via setting of the bit pvte.vol_trouble (see earlier description of this bit). This is also the case if an ESD occurs after a system crash at which time processes had marks in this table.

The segment mover marks two volumes at a time in this way.

The PVT hold table can be located, for crash analysis and debugging purposes, from the sppointer sst.pvthtp.

SECTION XIV

OPERATIONS OF RING-0 VOLUME MANAGEMENT

ACCEPTANCE OF PHYSICAL VOLUMES

The acceptance of physical volumes is the most fundamental and important operation of ring zero volume management. This service is provided for ring 1 volume management, which controls the operator and cross-process interface, at the time that the latter wishes to make a logical volume available for use. All of the physical volumes in a logical volume are accepted by ring 1 volume management before the logical volume is declared to be mounted (entered in the LVT). The main procedure of volume acceptance is accept_fs_disk.

Physical volumes are accepted by calling initializer_gate_$accept_fs_disk, with the PVT index of the drive on which the physical volume to be accepted is mounted. The ring 1 volume management and registration package ensures that the volume on the drive is the correct one requested by the operator or requesting processes. Ring zero volume management assumes that it is correct, and derives all data from the label of that volume. The RPV is accepted in a special fashion during collection 2 of bootloading; the operator, by issuing the BOOT command, and by use of the ROOT CONFIG card, has assured that the drive described by that card is the legitimate RPV. Thus, this physical volume is accepted automatically in ring zero without having been validated by ring 1.

The essence of physical volume acceptance is to initialize the PVTE for the drive on which the volume being accepted is mounted with data from the label, VTOC header, and volume map of that volume, and mark the PVTE as belonging to a volume in use. This latter step is the last step. Thus, there are no race conditions in determination of whether or not this volume is actually accepted. Since segment creation is driven off the logical volume table, and initiation checks there as well, it is only in the case of non-RPV volumes of the RLV that there is even an issue, for only in this case is there a LVT entry before all PVT entries are set up.

An auxiliary task of physical volume acceptance is to copy the volume map into the region allocated in the FSDCT for bit maps of volumes mounted on that drive. This function is performed in the procedure load_vol_map, which constructs a PTW-level abs-seg to read the volume map from the disk. This procedure also takes responsibility for reading the VTOC header (via the same abs-seg) and initializing PVTE parameters derived from the latter from it. In the case of the loading of the volume map of the RPV, page control activity is halted, via wiring, masking, and locking the page table lock, while the volume map is being copied. This is because the bit-map region of the FSDCT for the RPV will contain the bit-map of the hardcore partition at this time, and will actually be in use at that time. Although all pages of the supervisor should be withdrawn at that stage, and thus no activity on this bit map should take place during the copy, this policy assures that none in fact will take place. This policy dates from a time before all supervisor pages were prewithdrawn. The program load_vol_map also takes responsibility for filling in these PVTE parameters derived from the volume map.

It is also the responsibility of physical volume acceptance to determine if a physical volume needs salvaging, and call the physical volume salvager if so. A physical volume needs salvaging if it was in use, not properly shut down, and not salvaged since it was used. The volume map and VTOC may not be used validly unless this salvage is performed. Each time that a physical volume is accepted, the label is written out at the end of the acceptance procedure (via a call to fsout_vol), which sets label.time_map_updated to the current time. Each time that a physical volume is properly demounted (including shutdown), the label is written out, but this time, setting both label.time_map_updated and label.time_unmounted to the current time. Thus, if an attempt is made to accept a physical volume for which the value of label.time_map_updated and the value of label.time_unmounted are not equal, then this volume was not properly shut down. If, however, the volume has been salvaged since it was last used, it need not be salvaged again. The volume salvager writes out the label with label.time_map_updated and label.time_salvaged equal to the current time. The equality of these two label fields implies the completion of a volume salvage since last use. The procedure accept_fs_disk makes these checks for all volumes except the RPV; init_pvt, at RPV preacceptance time, makes these checks for the RPV.

The automatic salvaging of volumes during acceptance includes the repatriation of pages from that volume left on the paging device during the previous (or earlier) bootload. This is done in the case where the system has an unflushed paging device, and the physical volume salvager detects that the volume was not previously shut down, and exposed to the system's instance of the paging device. (see Sections IX and XV.)

PHYSICAL VOLUME DEMOUNTING

The demounting of physical volumes involves reversing all of the steps taken at acceptance time, and physically cycling down the disk drive on which a physical volume is mounted. Physical volume demounting is complicated by the fact that at the time that a physical volume is demounted, any number of processes may be using information on that physical volume, and may be depending upon its mounted and accepted status. The problems of demounting are thus two, the flushing of supervisor data bases of all information about the physical volume, and the stopping of processes that are using information on it, in a recoverable way.

The principal goal of demounting is the updating of all information on that physical volume with the latest copies of information resident in the AST, FSDCT, and in frames and records of main memory and paging device. This implies writing back all pages in main memory and paging device to their assigned addresses on that physical volume, and the updating of all VTOCEs for segments on that volume from the AST. These two steps are accomplished by deactivating all segments (see Sections II and IV) from that physical volume which are active at demount time. The VTOC manager's VTOC buffer segment must be flushed of all vtoce-parts from this volume, and all pendent I/O on it awaited. This step, clearly, is performed after the deactivation of all segments on the volume. The volume map, VTOC header, and label of the volume must be updated from the FSDCT and PVTE for the volume.

The procedure that coordinates demounting is demount_pv, also known as the demounter. The final stages of demounting, viz., updating the volume map, VTOC header, and volume label, are performed by fsout_vol, called from demount_pv. It is essential to realize that all volumes are demounted at shutdown time, both emergency and regular. There are only two differences between normal demount and shutdown demount.

1. At normal demount time, the drive containing the volume to be demounted is cycled down via a series of calls to the I/O Interfacer. At shutdown time, no drives are cycled down.

2. At normal demount time, segments are deactivated via a call to "deactivate," the normal segment control deactivation procedure. At shutdown time, explicit calls are made to pc$cleanup and update_vtoce (the two procedures at the heart of deactivation) to avoid dealing with possibly bad AST threads (and to allow deactivation of directories with active inferiors. Directories are all on the RLV, which cannot be demounted via normal demounting).

The demounter begins by turning on the bit pvte.being_demounted, and waiting for all processes engaged in multistep operations on this volume to finish. Turning on this bit, as explained below under "Demount Protection," prevents the inception of any new multistep operations on this volume after the time it is turned on. The demounter then locks the AST and deactivates (or, in the shutdown case, simulates deactivation of) all segments on this volume. This deactivation is performed under the AST lock; all processes seeking to activate a segment check the bit pvte.being_demounted at such time as they acquire the AST lock. Thus, since no process except that of the demounter holds the AST lock at this deactivation time, any process except that of the demounter holds the AST lock at this deactivation time, any process attempting to activate a segment, that did not succeed in fully activating it before the demounter acquired the AST lock, will acquire the AST lock after the demounter, and thus find the bit pvte.being_demounted _on_, and fail to activate the segment. Therefore, the deactivation of all segments on the volume is total and irreversible; it deactivates all segments that were active when it acquired the lock, and no segments (on that volume) will be activated after it releases it. The deactivation purges all data relevant to the volume being demounted from the AST and from page control, and makes the copies of all segments on the disk, and all VTOCEs accurate. This is what is normally done by deactivation (see Section IV); it is simply being performed here for all active segments on the volume.

The second phase of the demounter is the cessation of VTOC I/O activity for the volume. This begins by setting the bit pvte.being_demounted2, which prevents the inception of any VTOC I/O activity for the volume not already under way. As the deactivation phase of demounting starts a great deal of VTOC I/O activity for the volume, which does not complete in that phase, this phase must follow the deactivation phase. A call is made to the VTOC manager (vtoc_man$cleanup_pv, see Section III) to await all I/O in progress for vtoce-parts of this volume, and make a final attempt at flushing "hot" vtoce-part buffers (those that have suffered write errors). Before this call returns, all data relevant to the physical volume being demounted will have been flushed by the VTOC manager from its data bases. This call involves the VTOC manager locking its VTOC buffer lock. All other calls to the VTOC manager check the bit pvte.being_demounted2 under the protection of this lock, and return an error code (error_table_$pvid_not_found) if the PVTE of a volume specified to it has it on. Therefore, all VTOC I/O operations underway at the time the demounter acquires the VTOC buffer lock will be awaited to completion by the demounter, and, since any potential operation not under way by then will acquire the lock after the demounter and find pvte.being_demounted2 on, no new operation may be started after the demounter has released the lock. Therefore, the purge of information about the volume is total and irreversible; all VTOC I/O activity is complete for the volume, and no new activity may be started.

The third phase of demounting is performed by fsout_vol, which, in general, updates labels on disks. In the case of a demount, all parameters in the volume map (including the bit map itself), and the VTOC header are updated as well. The cells label.time_unmounted and label.time_map_updated are set to the same value (the current time), which indicates to the next attempt to accept this volume that it was successfully shut down. These policies are explained under "Physical Volume Acceptance" in this section. Once the label has been written out, the parameters in the PVTE for the volume's PVID to fail, and allowing reuse of the drive for acceptance of a (probably different) physical volume.

The final phase of demounting, which is not performed at shutdown time, is the cycling down of the drive on which the volume being demounted is mounted. This is performed via a "hardcore" attachment to the I/O interfacer to issue an "unload" command to the drive. An attachment is made, for the demounter process, via direct calls to the I/O interfacer. The resource control program (RCP) is not involved in any way. A workspace segment is set up by the I/O interfacer, and the procedure fs_unload_disk_interrupt is set up as the interrupt handler for the attachment. A connect is issued to the drive, to execute the "unload" command. The demounter sets a cell (pvt.rwun_pvtx) to the PVT index of the drive to which the "unload" command was issued before issuing the connect, and loops awaiting the zeroing of this cell. The interrupt-side program (fs_unload_disk_interrupt), after making a few checks, zeros this cell upon receipt of IOM status from the unload operation. The use of this single shared cell prohibits the demounting of several volumes in parallel; this fact is enforced by the restriction that only the initializer process can perform demounting. A single shared cell is used because the I/O interfacer provides no facility for its interrupt side to identify a device to a subsystem's interrupt handler in terms known to that subsystem. Thus, as there is no simple way to determine the PVT index of a drive to which an "unload" was issued at interrupt time, a single cell is used.

Demount Protection

The demounter poses to segment control the problem of the validity of PVT indices; a PVT index derived via search of the PVT for a given PVID is valid if and only if pvte.being_demounted was not on (volume was not being demounted) at the exact instant that the PVID was found in the PVT, and remains valid only as long as this is so. By "valid," we mean that use of this PVT index, by page control or VTOC management, will indeed result in a reference to the physical volume whose PVID was sought to determine this PVT index. Thus, a PVT index which was "validly" derived via PVT search can become invalid instantaneously as another process executes the demounter. Thus, without further mechanism, PVT indices would be useless, as they could be invalidated at any time. Mechanisms therefore exist to implement demount protection, via which processes can either ensure or determine the validity of PVT indices at any time.

The simplest of these mechanisms is the "unitary operation" facility provided by the VTOC manager. This can be used by any function that involves only a single interaction with the volume, and that interaction must be via the VTOC manager. Such an operation is the reading of "VTOC attributes" (see Section IV). A single call to the VTOC manager is adequate to supply such information. Another is the allocation of VTOCEs (see "Segment Creation," Section IV), for which exactly one call to the VTOC manager allocates and writes out a VTOCE. Such operations are said to be "unitary;" either the VTOC manager will succeed in performing them totally, or report that the physical volume is not mounted. These operations are made possible by supplying the PVID of a volume on which an interaction is necessary along with a possibly-valid PVT index for the drive on which that volume is (probably) mounted. This PVT index can be obtained via a call to get_pvtx$get_pvtx, which will make a perfunctory check for a being-demounted bit, and return the PVT index of the physical volume (if any) with that PVID. It is no matter that the volume may be demounted (pvte.being_demounted turned on, or fully demounted) after this search has been performed; the VTOC manager will check the PVTE specified by the PVT index supplied against the PVID supplied under the protection of the VTOC buffer lock before commencing any operation. If the PVID does not correspond, or the bit pvte.being_demounted2 is on (the point at which VTOC I/O request inceptions will no longer be honored), the request is refused. If the PVID corresponds, and the bit pvte.being_demounted2 is not on, the demounter cannot proceed, or even turn on this bit, until it acquires the VTOC buffer lock (see the preceding discussion) and cannot complete until the operation that is being requested here has finished (no vtoce-part buffer out_of_service bits are on).

If the operation being requested requires several vtoce-part I/Os, with intervening unlocks of the VTOC buffer lock, the operation may fail in an intermediate state. However, the design of the VTOC manager is such (see "VTOC Manager, General Policies," Section III) that no irreversible action will have been taken until all vtoce-parts are acquired in buffers under the protection of the VTOC buffer lock.

Another form of protection against demounting is provided to those procedures which operate under the protection of the AST lock. This specifically includes segment deactivation. Since the demounter must lock the AST in order to deactivate all segments, and, as shown above, no new segments can be activated after it has finished this activity, any PVT index obtained (under protection of the AST lock) from an ASTE is valid as long as the AST is locked to the process that obtained it, in the same locking. Any process that derives a PVT index by other means (PVT search, for example), while the AST is locked, is ensured of the validity of that PVT index for as long as the AST is locked, provided that pvte.being_demounted was not on at the time that it derived it (shortly before, or after, so long as the check is made with AST locked).

A similar form of protection is provided to the VTOC manager; if an operation is commenced under the protection of the VTOC buffer lock, and pvte.being_demounted2 was determined not to be on shortly after this lock was locked, the demounter cannot acquire the VTOC buffer lock as long as it is held by the current process, and thus the validity of the PVT indices so validated is ensured.

The most general form of demount protection is provided by the "demount protection brackets" implemented by the entries get_pvtx$hold_pvtx and get_pvtx$release_pvtx. Between a call to get_pvtx$hold_pvtx that does not fail (return an indication of demounted or demounting volume) and a call to get_pvtx$release_pvtx with the returned PVT index, by the same process, the volume specified by PVID to get_pvtx$hold_pvtx will not be demounted. The first call places, and the second call removes, a "mark" in the PVT hold table, specifying the process and the PVT index of the volume concerned. The demounter waits for all such marks for a given volume being demounted to be cleared from the PVT hold table as one of the first steps in demounting. To ensure that no new marks for a given physical volume are made once the demounter awaits the removal of all marks for that volume, the bit pvte.being_demounted is turned on before the demounter awaits the removal of these marks. The entry get_pvtx$hold_pvtx will return a failure indication if this bit is on before it makes its mark, and will remove its mark and return a failure indication if this bit is found on after it makes its mark. The entry get_pvtx$drain_pvtx is used by the demounter to await the removal of all marks relative to a given physical volume.

The demount protection brackets are used to "bracket" multistep interactions with a physical volume, protecting the entire interaction against the demounter. When such an operation has commenced, the demounter may not progress in such a way that would invalidate that operation until the operation is over. If a demount is in progress, such an operation may not even begin. Typical multistep volume interactions are truncation and deletion of segments. Truncation involves calling the VTOC manager to write back a VTOCE without certain addresses, followed by the depositing of these addresses (to the FSDCT). Should the volume concerned be demounted between the VTOCE write and the deposition, the deposition would address an invalid volume map in the FSDCT. Similarly, deletion of a segment involves truncation and freeing of a VTOCE; should the volume be demounted between the truncation and the freeing, a zero-length segment would appear on the volume the next time that the volume is accepted. Thus, these multistep operations must be bracketed by calls to get_pvtx$hold_pvtx and get_pvtx$release_pvtx, protecting the volume against demounting, and allowing the PVT index produced by the former to be used validly (without the protection of the AST lock).

Should a process encounter an ansychronous interruption (such as a "crawlout," process termination, or a crash followed by an emergency shutdown) at the time that a volume is "held" by the demount protection bracket mechanism, the procedure verify_lock (in the first two cases, or wired_shutdown in the third) will clear the mark from the PVT hold table, and schedule the volume for later salvage via the setting of the bit pvte.vol_trouble. This will cause later demounting of that volume to write out the label in such a way that it is volume-salvaged the next time that it is accepted.

The segment mover holds two volumes at a time, the two engaged in the segment move.

RING ZERO LOGICAL VOLUME MANAGEMENT

The logical volume is an instrument of convenience used to compensate for the inadequacy of a physical volume, in size, to hold an arbitrary number of segments. As such, the mounting of logical volumes is little more than the acceptance of several physical volumes, and the demounting of several physical volumes. Thus, the mounting and demounting of logical volumes is little more than the preparation and destruction of entries in the logical volume table describing the logical volume. Ring zero logical volume management also consists of the maintenance in the KST of each process of a small table of logical volume IDs (LVID's of private logical volumes mounted to that process.

Other than the setting of per-process (KST) and per-system (LVT) table entries, marking logical volumes as mounted or not mounted to the system or the calling user process, logical volume management provides only two services to the rest of the supervisor:

1. Answering the question of whether or not a given logical volume is mounted to the calling process. For a public logical volume, this is equivalent to whether or not it is mounted at all (to the system). For a private logical volume, it must be mounted to the system and attached to the invoking process. The procedure "mountedp" answers this question in general, given the LVID of a given volume. This code is duplicated in the segment fault handler for efficiency.

2. Providing the head of the PVT chain for a given logical volume, for the segment (VTOCE) creation function, described in Section IV. This service is provided by logical_volume_manager$lvtep, which returns a pointer to the appropriate LVTE, or null if that volume is not mounted to the system. This pointer may be invalidated at any time; the LVIDs of physical volumes as stored in the PVT are cross-checked by the segment creation function to account for this fact.

The logical volume table is manipulated without a lock; this is because only the mount/demount process (the Initializer) may modify it. Processes that search it are aware that the results of searching it may be instantaneously invalidated. Only in the case of segment creation is this an issue; at other times, subsequent calls to the VTOC manager will fail if physical volumes are demounted after a subsequently invalidated logical volume presence is deduced from the LVT. The LVT is managed by the program logical_volume_manager. The entries to add and delete logical volumes from the logical volume table (logical_volume_manager$add and logical_volume_manager$delete, respectively), are called by the ring 1 volume management package in the initializer process, which implements the operator interface, via the gate initializer_gate_. The "add" entry builds the LVTE from information supplied, and threads together the PVT chain of all existent PVTEs with an LVID equal to the LVID of the volume being added to the LVT.

The "delete" entry destroys this thread, and frees the LVTE. An entry exists (logical_volume_manager$add_pv) which adds a PVTE (and thus a physical volume) to an alreadymounted logical volume. This is used by the ring 1 volume management package when other physical volumes of the RLV than the RPV are accepted, at which time the RLV is already mounted, and at the time that new physical volumes are created and accepted while a logical volume is mounted.

The table in the KST (kst.lv) of private logical volume LVIDs is used to answer the question of whether or not a private logical volume is mounted to the process owning the KST. A call to private_logical_volume$connect, from the ring 1 Resource Control Package (RCP), adds an LVID to this table. Before this call is made, RCP validates the caller's access to the logical volume, and the fact that it is mounted to the system (at least immediately before the call is made). This call is made via the gate admin_gate_. The complementary call to private_logical_volume$disconnect removes an LVID from this table. At the time any segment on a private logical volume is initiated in a process, its index in this table is stored in its KST entry (kste.infcount, multiplexed because all directories, the only segments with nonzero inferior counts, are on the RLV, a public volume). At the time that an LVID is removed from a process' KST, a setfaults operation (setfaults$disconnect, see Section II) is performed on each known segment in this process on that logical volume. This causes the immediate revocation of access to that volume for the process, as the segment fault handler checks whether or not a logical volume is mounted to a process (defined) before honoring a segment fault on that volume for that process.

The entry private_logical_volume$lvx exists to answer the question as to whether a given LVID appears in the calling process' KST, i.e., is mounted to that process <u>given</u> that it is mounted to the system (as determined by logical_volume_manager$lvtep).

<u>BOOTSTRAPPING OF LOGICAL VOLUME HIERARCHY (THE RPV)</u>

The system must be booted to command level before the operator can issue commands to cause the acceptance of physical volumes and the mounting of logical volumes. However, the running of the operator software, and the loading of the system library segments into the hierarchy, involves directories in which to put them, and thus the existence of the root logical volume, before these commands can be issued. Thus, it would at first seem that the RLV must be mounted before the system comes up. Mounting of logical volumes automatically by ring zero is undesirable, as it requires that ring zero be informed of the location of these volumes via CONFIG cards, or various inflexible forms of contract based upon configurations during the last bootload. The responsibility of validating labels resets upon the ring 1 volume management package. Thus, the compromise is made that only one physical volume of the root logical volume must be present at bootload time; this volume is the RPV, and the description of its drive via the ROOT CONFIG card constitutes validation of the RPV pack as the RPV by operator. All of the directories needed by bootloading, that already exist, must be on this particular volume of the RLV. Furthermore, the segment used by cross-bootload ring 1 volume management (>disk_table) to specify the location of packs during the last bootload, must be available on this volume, as all volumes are assumed, by covenant with the operator, to assume their positions during the last bootload unless otherwise specified.

All of the directories so needed are either the root directory itself (>) or one of its immediate descendants (>dumps, >system_library_1, or >process_dir_dir.) Thus, by placing the cross-bootload disk configuration segment (>disk_table) in the root directory, the rule can be made that all immediate descendants of the root directory (segments or directories) must be allocated on the RPV. The segment creation function (see Section IV) carries out this policy; any segment or directory created off the root directory can only be created on the RPV. The segment mover will not move such segments off the RPV.

An implication of this policy is that the RLV must be mounted to the system (so that segment creations and segment faults may be honored upon it) while only the RPV is accepted. System initialization causes this to be the case by calling logical_volume_manager$add for the RLV at such a time during initialization that the RPV has been accepted. Ring zero has no notion of the completeness of volumes; any time that a call is made to logical_volume_manager$add, that volume becomes usable, and consists of all of the physical volumes in it accepted at that time. All segment creations will be restricted to those volumes. Thus, all segments created by initialization reside on the RPV.

RPV-only Directories

When the system arrives at ring 1 command level, the RPV is the only physical volume accepted, and the RLV the only logical volume mounted. In order to register other logical volumes, and check their labels, the logical volume registration data base must be present. Thus, the logical volume registration segments used by the ring 1 volume management and registration package must be on the RPV. Rather than put these segments in the root directory, a directory exists (>lv) which has the property, like the root directory, that all of its inferior segments are restricted to allocation upon the RPV. The bit dir.force_rpv in this directory's header (set by set_sons_lvid$set_rpv, see the following discussion), has the same effect upon the segment creation function as creation of an immediate descendant of the root directory.

One peculiarity in this policy exists. Segments created by bootloading in >system_library_1 are not bound to stay on the RPV, and may be subject to segment moving. If the next bootload, which generally deletes all segments in >system_library_1 that appear on the new bootload tape, finds such a segment, which has been segment-moved, it cannot delete it. Initialization renames it in order to load the new one, with a message from make_branches$delete indicative of this fact. Such segments may be deleted by the "ldelete" command by system maintenance personnel, when the system is fully up (the entire RLV accepted).

Cold Boot of the RPV

During a cold system boot, when there is no hierarchy at all, the system must arrive at ring 1 command level before any volume registration commands can be issued. The RPV must be fully initialized and registered before it can be used, but before the system comes up. Therefore, the program init_pvt, when it detects a cold boot situation, "registers" the RPV by generating an LVID and PVID for it based upon the clock value. The program init_empty_root is called in this case, which writes a valid label for the RPV, including in it information placed on the special-format PART cards used in such a circumstance (see the Multics Operators' Handbook, Order No. AM81). The volume map, VTOC header, and VTOC are initialized, using default parameters generated by init_empty_root. The program that initialized the volume map, VTOC header, and VTOC, init_vol_header_, is available in all rings (a deciduous segment, see Section VII), and is used by the ring 1 volume management package to initialize other volumes. It takes as an argument an entry variable, specifying a routine that is used to write to the pack.

The ring 1 volume registration package (at mdx$reregister) constructs the RPV's registration information (as well as the RLV's initial registration information) based upon the information generated by initialization in ring 0 and written to the RPV label.

SONS-LVID SETTING

The directory field dir.sons_lvid is the logical volume ID (LVID) of that logical volume on which all immediately inferior segments to that directory will be allocated; this value is used by the segment creation function to obtain (via logical_volume_manager$lvtep) the head of the PVT chain of that logical volume. This value is also "inherited" as a son's-LVID by all directories created inferior to this directory. In all cases except the case of the creation of a master directory, this quantity is in fact inherited by the directory control directory creation primitive. In the case of a master directory, this value is specified by master directory control.

The sons-LVID of a directory may be changed dynamically, via the set_sons_lvid command (see the Multics Administrators' Manual -- System Administrator, Order No. AK50) if that directory has no immediately inferior segments (but *may* have inferior directories). This primitive accesses the program set_sons_lvid in ring zero via the gate hphcs_. This program simply changes the sons-LVID field of the directory, and marks it as (implicitly) a master directory, marking the ASTE as well as necessary. This feature is useful to cause process directory segments to be allocated on logical volumes other than the RPV; bootload re-creates >process_dir_dir each bootload, after renaming the old one. Thus, >process_dir_dir (and the initializer's process directory, >pdd>!zzzzzzbBBBBBB), have a son's logical volume of the RLV. Setting the son's-LVID of >process_dir_dir to some other logical volume after the system is up causes newly created process directories to inherit that son's-LVID, rather than the RLV.

RPV-ONLY DIRECTORY SETTING

The program set_sons_lvid also includes an entry, set_sons_lvid$set_rpv, to set the RPV-only bit (dir.rpv_only) for some directory whose son's-LVID is already the root logical volume (RLV). This facility, accessed through the gate hphcs_, is used by the ring 1 volume management package, to force volume registration files in the directory >lv to be on the RPV, so that they will be available in the ring 1 operator environment at bootload time, whether or not any other physical volumes of the RLV have been accepted.

DISK TABLE LOCATION SETTING

A facility exists to store the VTOC index (in the RPV VTOC) and unique ID of the segment >disk_table in the label of the RPV. The ring-0 primitive set_disk_table_loc is called (via the gate initializer_gate_) at the time the ring 1 volume management package is initialized to set this information. It obtains it from the branch of that segment, and stores it via reading and writing the RPV label. This information is placed there for the use of an unimplemented facility whereby BOS SAVE would be able to determine the location of physical volumes by reading the disk table, rather than receiving volume location specifications on individual request lines.

EXPLICIT DISK READING, WRITING, AND TESTING (read_disk)

Volume management provides a utility program (read_disk), which, given a (guaranteed valid) PVT index, record number, and data buffer, reads or writes that record from/to that data buffer. This is accomplished via the use of a PTW-level abs-seg (see Section VII), rdisk_seg. In the reading case (read_disk), a live device address, the record address desired, is placed as a "disk devadd" (see Section VI) in the single used PTW for this segment, and data copied from rdisk_seg to the caller's buffer. In the write case, a nulled "disk devadd" describing the record described is placed in the PTW, and the data copied from the caller's buffer to rdisk_seg. The nulled address prevents the old data from being read in in order to page the new data out. This is relevant performance of this primitive in cases where it is used in a loop (such as volume initialization). After either call, pc$cleanup is used to force the page of rdisk_seg out of main memory (see Section IX, "Deactivation Service"), in the write_disk case, causing the actual write, and guaranteeing its completion to the caller. The PVT index supplied, in either case, is placed in the ASTE for rdisk_seg before the reference to this abs-seg. This selects the drive to be addressed.

The primitives read_disk and write_disk call a special entry in the disk DIM (disk_control$test_drive) to determine if a drive is patently inoperable before attempting to use it via abs-seg (paging) I/O, which would generate disk DIM and page control error messages in that case. This special entry is used by turning on the bit pvte.testing in the PVTE for the drive concerned, calling it, and looping on the bit pvte.testing, waiting for it to be turned off by the disk DIM interrupt side. The DIM issues a "RQS" (Request Status) operation on behalf of this entry, and sets the bit pvte.device_inoperative to report the outcome of this operation. The bit pvte.testing is turned off once pvte.device_inoperative is set appropriately. If this test indicates an inoperative drive, read_disk and write_disk return an appropriate error code, and do not attempt paging I/O on the volume. This testing function is also available explicitly via the entry read_disk$test_drive.

The read_disk and write_disk entry points are used by acceptance and physical volume demounting to read and write labels, VTOC headers, and volume maps (although load_vol_map uses its own abs-segs). These facilities are also available to the ring 1 volume management package via initializer_gate_$read_disk and initializer_gate_$write_disk, to verify labels, and perform volume initializations. As there is only one ASTE for the abs-seg rdisk_seg, all of these activities are confined to the initializer process, or the process performing emergency shutdown.

SECTION XV

INTERACTION OF THE PHYSICAL VOLUME SALVAGER WITH THE STORAGE SYSTEM

This section describes the actions performed by the physical volume salvager as they are relevant to the actions performed and assumptions made by volume management, segment control, and page control. It does not attempt to explain the internal organization of the physical volume salvager, its interface with the rest of the Multics salvager subsystems, or the interpretation of its printed diagnostics. For these details, see the Multics Storage System Salvager PLM, Order No. AN62, and the Multics Operators' Handbook, Order No. AM81.

The physical volume salvager is invoked upon a single physical volume. It may be invoked either by explicit operator command (the salvage_vol Initializer command, see the MOH), or automatically by physical volume acceptance (see Section XIV) if the latter determines that the volume being accepted was not properly demounted during its last use. The physical volume salvager inspects and modifies the label, volume map, VTOC header, and VTOC of a physical volume, using abs-seg I/O. The tasks of the physical volume salvager are two:

1. To make valid a set of assumptions about the VTOC and volume map of a physical volume, on which the proper operation of segment control, page control, and volume management depend. These assumptions are detailed below.

2. To detect and correct random and unexplained damage, due to hardware or software failure, to the VTOC and volume header of a volume.

The first objective repairs "damage" to a physical volume that occurs any time use of that volume is stopped (by a crash, or drive failure, for instance), without proper demounting as detailed in Section XIV. For instance, any volume whose use is stopped without proper demounting will contain an invalid volume map, for no attempt is made to update the volume map until demount time. Such a volume may contain an invalid VTOCE free list, as VTOCEs are freed and disk requests are executed in not necessarily the same order.

The second objective repairs damage that cannot come about simply by improper shutdown; there is no way that the storage system will allow inconsistent states to exist wherein reused addresses appear. If a reused address appears in a VTOC, it is due to undetected hardware or software failure. This is also the case if the static parameters of the volume map, for example, become inconsistent with the volume label. No accounting can be made for such damage, nor can the actual "correct" state ever be exactly determined. Such damage, which is rare, must be "corrected" to satisfy the primary goal of the physical volume salvager, the validation of storage system assumptions.

ASSUMPTIONS MADE VALID BY THE PHYSICAL VOLUME SALVAGER

The following are the assumptions about the state of a volume, which may not be true if the volume is not properly shut down, which are made true by the physical volume salvager:

1. The current-length (vtoce.csl) of each segment, in its VTOCE, describes the 1-relative page number of the highest nonnull address in the file map.

2. The records-used (vtoce.records) of each segment, in its VTOCE, is the number of nonnull addresses in its file map. Like vtoce.csl, this will not be true for active segments that suffered page creation or deletion while active and received VTOCE updates before use of the volume was interrupted.

3. The volume-map has a "0"b for every record address cited in a VTOCE on this volume, and a "1"b for every other address in the paging region.

4. The volume map has the correct number of "0"b bits in the volume map, when (3) is true.

5. Every free (vtoce.uid = "0"b) VTOCE is part of a consistent, nonlooped chain, whose head is kept in vtoc_header.first_free_vtocx. The end of the chain is -1.

6. The cell vtoc_header.n_free_vtoce describes the number of VTOCEs in the chain as described by (5).

If these assumptions are not true for a volume that is accepted, segment control, page control, and volume management will malfunction. These assumptions are always true for a volume that has been demounted properly. Thus, the acceptance of any volume that has not been properly demounted implies a volume salvage to force these assumptions true.

The physical volume salvager reports any deviance from assumptions 1 and 2. These reports may be taken as cues to the damaging of active segments by improper shutdown.

FORMS OF DAMAGE CORRECTED BY THE PHYSICAL VOLUME SALVAGER

The following further forms of damage to physical volumes are corrected, via various assumptions, by the physical volume salvager. Such damage cannot result simply from improper or non-existent shutdown. Software or hardware damage to the volume is a prerequisite.

1. An address appearing in more than one VTOCE. If one page so affected is a page of a directory and one is not, the directory is awarded the page. Otherwise, zeros (via a null address) are assigned to both pages.

2. Inconsistent maximum length (vtoce.msl less than vtoce.csl). It is set to current-length.

3. Addresses not on the legal boundaries of the paging region of the volume. They are replaced in the VTOCE file map by null addresses.

4. Inconsistency of the global volume map parameters (there were software problems creating these inconsistencies in release 4.0). They are corrected on these assumption that these known software problems (in the disk rebuilder) caused them.

OTHER VOLUME SALVAGER ACTIONS

The running of the physical volume salvager is primarily a walk through the VTOC of the physical volume being salvaged, recreating the volume map and checking individual VTOCEs. In the case where a volume that has not been properly shut down is being salvaged, and the system has an unflushed paging device, the physical volume salvager makes a call to page control (pc$flush_seg_old_pd) for each VTOCE processed, in order to repatriate pages of the segment owning the VTOCE trapped (at crash time) on the paging device. This service of page control, the post-crash PD flush, is fully described in Section IX. This service of page control is passed the file map region of the VTOCE as a parameter; page control may place disk record addresses in it, in the case where the post-crash flush resurrects addresses. The physical volume salvager's checking of current length and records-used is postponed until such resurrection has been performed.

The physical volume salvager terminates by setting the label variables label.time_map_updated and label.time_salvaged to the same value, the current time. This will cause subsequent acceptance of the volume to realize that the volume is consistent, i.e., satisfies the conditions above, and need not be salvaged again. See "Physical Volume Acceptance" in Section XIV.

THE DISK REBUILDER

The disk rebuilder is a special version of the volume salvager that copies one physical volume onto another, reassigning address and reallocating partitions. The disk rebuilder is invoked via the "rebuild_disk" operator command, described in the Multics Operators' Handbook, Order No. AM81. The disk rebuilder copies the contents of partitions, and copies VTOCEs from the source physical volume to the target. Addresses on the target volume are allocated by the rebuilder, and the contents of pages of segments copied from the target volume to the addresses so allocated via the explicit disk reading and writing mechanism described in Section XIV.

The disk rebuilder updates the VTOCEs of all active segments by searching the AST for each segment, and performing the page-control deactivation service (see Section IX) and a VTOCE update (See Section IV) for each segment found active before it is copied. This updates the VTOCE and segment pages on the disk.

The shutdown state and label times of the disk being copied are falsified by the disk rebuilder in the case where an accepted physical volume is being copied. Were this not the case, a volume being so copied would appear to have crashed during the middle of a disk rebuild.

ASSUMPTIONS NOT CHECKED BY THE VOLUME SALVAGER

The following assumptions about the storage system hierarchy must be true in order to ensure correct operation of the system. They can become invalid by interruption of operation or use of a physical volume. However, since all of these assumptions take great expenditures of real time to be made true, the system is prepared to operate without their being true. The adverse effect which will result is detailed in each case.

1. All directories have valid threads and formats. This assumptions is made valid by a full run of the hierarchy salvager. If a directory is encountered with any of various invalid threads and formats during normal operation, a crawlout will occur. The online salvager will salvage that directory, and cause this assumption to be valid.

2. Every VTOCE that is designated by a PVID-VTOC index pair in a segment or directory branch in fact is in use, and indeed is the VTOCE for that segment or directory (vtoce.uid must be the same as entry.uid). A segment or directory for which this is not true is said to suffer a connection failure (See Sections II and IV). Any primitive which accesses a VTOCE, from a branch, is prepared for this occurrence, and will return error_table_$vtoce_connection_fail. Such "segments" may be deleted, but not activated. A full run of the hierarchy salvager in "check_vtoce" mode (see the MOH) will detect and delete all such branches.

3. For every VTOCE, there must be a branch which, via a PVID-VTOC index pair, designates this VTOCE. A VTOCE for which this is not true is said to suffer reverse connection failure. The effect of this problem is wasted VTOCEs, and wasted disk records (the records designated by the file maps in such VTOCEs), as the "segments" they describe are not in any way accessible. The tool sweep_pv (see the Multics Operators' Handbook, Order No. AM81), invoked for "garbage collection and deletion", reports and deletes such "orphan" VTOCEs (See "Special Services for sweep_pv", Section IV).

4. The "quota used" cell of every directory must contain a number equal to the sum of the "records-used" fields of all immediately inferior segments, and of the "quota used" cells of all immediately inferior directories that do not have their own quota accounts. This is the definition of "quota used". When this is not so, users experience negative used figures and other false used figures, being charged for nonexistent pages or not being charged for existent pages. A full run of the hierarchy salvager in "check_vtoce" mode remedies this situation. Similarly for directory quota.

The assumptions 1, 2, and 4 are made valid by the hierarchy salvager for all directories critical to the booting of the system, during bootload, if it was determined that the system was not shut down properly (and hence the RPV, and thus the RLV, on which all directories exists) during the previous bootload. It is only in this case that these anomalies can occur. The system forces these assumptions to be true for these critical directories by automatic invocation of the hierarchy salvager during bootload and system startup.

SECTION XVI

SCENARIOS

This section gives two scenarios of typical operations in the storage system, showing who calls what and how, and what data is affected. The handling of a typical segment fault and a typical page fault are detailed in this way. These sequences are intended to be typical, not canonical.

A SEGMENT FAULT

We will consider a segment fault on >udd>x>y>z. >udd>x>y is known with a segment number of 243, and >udd>x>y>z with 244. The segment >udd>x>y>z is described by VTOCE 2045 on physical volume pub01, which is mounted on the drive whose PVT index is seven. The current length of this segment is 100K. >, >udd, and >udd>x are active, and >udd>x>y and >udd>x>y>z are not.

A reference to 244¦14 is made by the processor. A directed fault 0 occurs.

The module "fim" is invoked, recognizes this directed fault as a segment fault, and invokes the segment fault handler, seg_fault.

The segment fault handler determines that indeed there is no SDW for segment 244, and it is not a process stack.

seg_fault calls sum$getbranch_root_my with the pointer 244¦14, hoping to obtain a pointer to its branch.

sum$getbranch_root_my inspects the KST entry for segment 244, determining from kste.entryp that its branch resides (as seen by this process) at 243¦5730, in >udd>x>y. sum$getbranch_root_my calls lock$dir_lock_read to lock this directory to validate the branch.

lock$dir_lock_read tries to touch >udd>x>y, but takes a segment fault. A directed fault 0 occurs.

The module "fim" is invoked recursively, recognizes the segment fault, and invokes the segment fault handler recursively.

The segment fault handler processes the segment fault on 243¦10, performing the actions now being recursively described.

The fim is returned to, and restarts the reference made by lock$dir_lock_read.

lock$dir_lock_read places an entry in the dirlock_table, locking >udd>x>y to this process.

sum$getbranch_root_my calls validate_entryp to ensure that 243¦5730 is still the branch for z.

sum$getbranch_root_my returns the pointer 243¦5730 to seg_fault, with >udd>x>y locked to this process.

seg_fault checks the time in that branch against the time in the KSTE to ensure that the access calculated at initiate time is still valid.

seg_fault calls activate with the pointer to z's branch, 243¦5730, to receive an AST entry pointer with the AST locked.

activate copies critical information out of the branch at 243¦5730 into its stack, and locks the AST, in order to determine if >udd>x>y>z is active.

activate calls search_ast with the UID of z to search the AST for z. search_ast replies that z could not be found, and is thus not active. activate unlocks the AST.

activate calls get_pvtx with the physical volume ID of volume pub01 (from the branch at 243¦5730), to get a PVT index. This program returns "7". This number can be invalidated at any time.

activate calls vtoc_man$get_vtoce with the PVT index 7, the PVID of pub01, and the VTOC index of z's VTOCE, 2045, the latter two items culled from the branch at 243¦5730. The first vtoce-part is requested.

vtoc_man$get_vtoce locks the VTOC buffer lock, and calls GET_BUFFERS_READ to see if PVT index 7, VTOCE 2045, first vtoce-part is present. It is found by SEARCH in vtoce-part buffer 33. It is copied out to activate's stack frame, and the VTOC buffer lock is unlocked.

activate sees that z is longer than 96K, and that the second vtoce-part will be required to get the file map. Activate calls vtoc_man$get_vtoce asking for PVT index 7, VTOCE 2045, second vtoce-part.

vtoc_man$get_vtoce locks the VTOC buffer lock, and calls GET_BUFFERS_READ to find the second vtoce-part of PVT index 7, VTOCE 2045. It is not found. A vtoce-part buffer (15) is pre-empted from PVT index 6, VTOCE 1011, third vtoce-part, and the disk DIM is called to read the second vtoce-part of PVT index 7, VTOCE 2045 into it.

vtoc_man unlocks the VTOC buffer lock, having set buffer 15 out-of-service, and calls pxss$wait to wait for the event 333000000015.

The disk DIM interrupt side calls vtoc_interrupt with the main memory address of VTOC buffer 15. The out-of-service bit is turned off, and pxss$notify is called to notify the event 333000000015.

pxss$wait returns to vtoc_man, which relocks the VTOC buffer lock, and copies buffer 15 into activate's stack frame.

vtoc_man unlocks the VTOC buffer lock, and returns to activate.

activate locates the ASTE for segment 243 in this process, >udd>x>y. It is at 17¦20444.

activate calls get_aste to obtain a 256-word AST entry to hold the segment z.

get_aste inspects the first ASTE on the 256K used list. It is for >udd>m>joe>bill.list, which has 12 pages in main memory.

get_aste inspects the second ASTE on the list. It is for >udd>m>cp>temp, which has no pages in main memory, and has had none come in since get_aste last saw this ASTE. It will be deactivated.

get_aste calls deactivate, passing it 17¦24644, the address of the AST entry of >udd>m>cp>temp.

deactivate calls setfaults to destroy all SDWs for >udd>m>cp>temp.

setfaults runs down the trailer list for >udd>m>cp>temp, locating all SDWs and, accessing the descriptor segments of various processes via an SDW-level abs-seg, removes these SDWs. Setfaults calls to clear the system's associative memories.

deactivate calls pc$cleanup to get all pages out of main memory.

pc$cleanup locks the page tablelock, and finds that no pages are in main memory for >udd>m>cp>temp. It unlocks the page table lock.

deactivate calls update_vtoce to update VTOCE 2311, PVT index 6 which, as determined from aste.pvtx and aste.vtocx in the ASTE at 17¦24644, are the PVT index and VTOC index of the VTOCE for >udd>m>cp>temp.

update_vtoce finds that >udd>m>cp>temp is 138K long, and no vtoce-parts will have to be read.

update_vtoce calls pc$get_file_map, passing it the AST address 17¦24644, to get a copy of the AST, with definitive information.

pc$get_file_map locks the page table lock, constructs a valid copy of that ASTE in its stack, unlocks the page table lock, and copies it out to update_vtoce's stack frame. This includes the file map.

update_vtoce constructs an image of VTOCE 2311 on PVT index 6, first two vtoce-parts, from the information returned by pc$get_file_map.

update_vtoce calls vtoc_man$put_vtoce with a zero as PVID, the PVT index 6, the VTOC index 2311, the image of the first two vtoce-parts of this VTOCE, and a request to write out these vtoce-parts.

vtoc_man$put_vtoce locks the VTOC buffer lock, and searches for buffers containing these vtoce-parts. None do. GET_BUFFERS_WRITE causes two other vtoce-parts to be preempted, and returns to vtoc_man$put_vtoce their indices, 23 and 16.

vtoc_man$put_vtoce copies the two vtoce-parts supplied by update_vtoce into buffers 23 and 16 respectively, setting these buffers out-of-service.

vtoc_man$put_vtoce calls the disk DIM to start writing the two buffers 23 and 16, and unlocks the VTOC buffer lock.

vtoc_man$put_vtoce returns to update_vtoce with the I/O still in progress.

update_vtoce determines that no nulled addresses were culled by pc$get_file_map, and VTOCE I/O completion will not have to be awaited.

update_vtoce returns to deactivate, having updated VTOCE 2311 on PVT index 6.

deactivate calls put_aste to free the ASTE at 17¦24644. It is moved to the head of the used list.

get_aste returns the ASTE at 17¦24644, now free, to activate.

activate connects the ASTE for >udd>x>y (17¦20444) with the ASTE to be used for >udd>x>y>z at 17¦24644.

activate calls pc$fill_page_table to fill in the ASTE's page table with information in the VTOCE (2045) on PVT index 7 which has been read in.

pc$fill_page_table converts the formats of the device addresses, and initializes the PTWs in this ASTE. A check is made for reused addresses.

The disk DIM interrupt side calls vtoc_interrupt, placing VTOC buffer 23 no longer out-of-service.

activate fills in the activation attributes of >udd>x>y>z into the ASTE at 17¦24644, along with other information.

activate returns to seg_fault with the AST locked, returning the ASTE pointer 17¦24644 for >udd>x>y>z.

seg_fault sets the encacheability state of >udd>x>y>z to "one process, reading and writing, encacheable".

seg_fault constructs an SDW for >udd>x>y>z, and places it at slot no. 244 in this process' descriptor segment.

seg_fault constructs a trailer entry in str_seg for this descriptor, giving the number 244 and the ASTE offset of this process' descriptor segment.

seg_fault unlocks the AST.

The disk DIM interrupt side calls vtoc_interrupt, placing VTOC buffer 16 no longer out-of-service.

seg_fault calls lock$dir_unlock to unlock >udd>x>y.

seg_fault returns to the fim.

The fim restarts the machine conditions for the segment fault.

The process proceeds, and the segment fault has been resolved.

A PAGE FAULT, IN PAGE MULTILEVEL

Having resolved a segment fault on >udd>x>y>z, our user process next attempts to access location 14.

The appending unit finds PTW 0 for segment 244 (at 17¦14660) to have ptw.df off. A directed fault 1 occurs.

The page fault handler, page_fault, is invoked. It saves all registers and machine state at pds$page_fault_data. It sets up a stack frame on the PRDS.

page_fault attempts to lock the page table lock, but finds it locked.

page_fault branches to pxss$ptl_wait to wait for the page table lock.

pxss$ptl_wait locks the traffic controller lock. It finds that the page table lock is indeed still locked. The cell sst.ptl_wait_ct is incremented.

This process is made to wait for the "PTL Event".

The process which had been holding the page table lock unlocks it, but notices sst.ptl_wait_ct nonzero.

The process which had been holding the page tablelock calls pxss$page_notify to notify the "PTL Event".

Our process resumes. pxss$ptl_wait returns to page_fault$wait_return, and the page fault is restarted.

The appending unit finds PTW 0 for segment 244 (at 17¦24660) to have ptw.df off, all over again. A directed fault 1 occurs.

The page fault handler, page_fault, is invoked. It saves all registers and machine state at pds$page_fault_data. It sets up a stack frame on the PRDS.

page_fault attempts to lock the page table lock, and succeeds.

page_fault calls pd_util$check_pd_free_and_update to see if the paging device needs housekeeping.

pd_util$check_pd_free_and_update determines that the PDMAP has been written out in the last second, and will not write it out.

pd_util$check_pd_free_and_update sees that only 8 PD records are free. 10 must be free or being freed.

pd_util$check_pd_free_and_update walks down the PD used list to find entries to free. Eight PD records are skipped, and the one at 17¦6440 is found. It is found to describe a PTW at 17¦15262, which describes a page in main memory. It is not a good candidate for replacement.

The next PD record on the used list, at 17¦6224, similarly describes a page in main memory, and is skipped.

The next PD record on the used list, at 17¦6030, describes a page not in main memory. It is not "PD Mod" (pdme.mod = "0"b).

pd_util$check_pd_free_and_update evicts this page from the paging device, taking the disk device address at 17¦6031 and placing it in the PTW (17¦17327) pointed at by pdme.ptwp at 17¦6032.

pd_util$check_pd_free_and_update calls pd_delete_ in the same program to put the PDME at 17¦6030 in the used list as free. The count of free PDMEs is now 9.

pd_util$check_pd_free_and_update considers the next PDME in the list, the one at 17¦6204. It describes a page whose PTW is at 17¦22137, and a nulled disk address (401512) on PVT index 6.

pd_util$check_pd_free_and_update calls rws_ in the same program, to start an RWS for the PD record whose PDME is at 17¦6204.

rws_ calls page_fault$find_core to find a main memory frame in which to perform the RWS.

page_fault$find_core picks up sst.usedp, which has the value 1550.

The core map entry at 17¦1550 is inspected. It describes a page whose PTW at 17¦21532 is modified with respect to paging device or disk (ptw.phm = "1"b). This page is not acceptable for eviction.

The core map entry at 17¦1304 is pointed at by cme.fp of the one at 17¦1550. It describes a page whose PTW at 17¦16120 describes a pure page which was recently used. This page is not acceptable for eviction.

The core map entry at 17¦1340 is pointed at by cme.fp of the one at 17¦1304. It describes a page whose PTW at 17¦17172 indicates that this page is pure, and not "not-yet on paging device".

The access to the page whose PTW is at 17¦17172 is turned off, by turning off the directed fault bit in that PTW, and clearing the system's associative memories, and clearing the caches of that page's words.

find_core calls page_fault$cleanup_page, which puts the PD address in the CME at 17¦1340 back in the PTW at 17¦17172, and adjusts the AST entry for this segment at 17¦17154. The CME at 17¦1340 is freed.

find_core returns the CME at 17¦1340 to rws_.

rws_ threads out the PDME at 17¦6204 and the CME at 17¦1340, and cross-relates them to indicate the read cycle of an RWS.

rws_ calls the bulk store DIM to read PD record 41 (whose PDME is at 17¦6204) into location 700000 in main memory whose CME is at 17¦1340. The bulk store DIM starts this read.

rws_ returns to pd_util$check_pd_free_and_update, who now notes that there are 10 PD records free or being freed.

pd_util$check_pd_free_and_update notes that there are incomplete RWS reads, and calls the bulk store DIM in a loop until there are none.

At one of these times, the bulk store DIM notices status for the read into location 700000, and calls page_fault$done_ with the address 700000, and an error code of zero.

page_fault$done_ locates the CME at 17¦1340, and inspects it, noting that an RWS read was in progress. The routine read_write_sequence in done_ is invoked.

page_fault$done_ acknowledges the RWS read completion, and indicates in the CME at 17¦1340 that an RWS write is in progress. The disk DIM is called (via device_control$dev_write) to write the address 001512 on PVT index 6 from location 700000.

page_fault$done_ returns to the bulk store DIM.

The bulk store DIM returns to pd_util$check_pd_free_and_update.

pd_util$check_pd_free_and_update notices that there are no more RWS reads outstanding, and returns to the page fault handler.

The page fault handler inspects the SCU data at pds$page_fault_data, and determines that this is not a descriptor segment page fault.

The page fault handler locates the SDW for segment 244, implicated in the machine conditions, and subsequently its page table at 17¦24660 and its ASTE at 17¦24644.

The page fault handler inspects the PTW at 17¦24660, and determines that indeed a page fault situation exists.

The page fault handler calls read_page, passing it the PTW address 24660 (relative to the SST), requesting the allocation of a main memory frame and subsequent readin of a page.

read_page checks that a nonnull address exists in the PTW at 17¦24660. It is address 002167 on PVT index 5, which is, by virtue of its format, nonnull and nonnulled (live). Thus, no quota check or allocation will be necessary.

read_page calls find_core to get a main memory frame into which to read that address.

find_core inspects the first CME on the main memory used list. This CME, at 17¦2200, was pointed to by cme.fp of the one at 17¦1340, which is now threaded out as an RWS is in progress there.

The CME at 17¦2220 describes a page whose PTW (at 17¦14140) indicates ptw.nypd, requiring allocation to the paging device. This page is not suitable for eviction. Its cme.fp pointer designates the CME at 17¦2214.

The CME at 17¦2214 designates a page that is neither modified nor "not yet on the paging device". Its PTW is at 17¦15150, and it will be evicted.

The access to the page whose PTW is at 17¦15150 is turned off, by turning off the directed fault bit in that PTW, and clearing the system's associative memories, and the caches of that page.

find_core calls page_fault$cleanup_page, which puts the PD address in the CME at 17¦2214 back in the PTW at 17¦15150. The AST entry at 17¦15130 is adjusted appropriately, and the CME at 17¦2214 is freed.

find_core returns the CME at 17¦2214 to read_page.

read_page sets the CME at 17¦2214 to indicate a page out-of-service on a read. The disk address 002167 is copied from the PTW at 17¦24660 to this CME. It is threaded out of the core used list. The main memory address, 230000 is placed in the PTW at 17¦24660, but ptw.df is still off.

The disk DIM is invoked, via device_control$dev_read, to read the record 002167 from the disk on PVT index 5 into location 230000 in main memory, the location described by the CME at 17¦2214. The read is started.

read_page returns to the page_fault_handler, informing it that waiting will be necessary.

The page fault handler calls claim_mod_core to start all I/Os that were skipped by find_core during this page fault.

claim_mod_core inspects sst.wusedp, which describes the CME at 17¦1550. This page was skipped because it needed writing.

claim_mod_core calls write_page, passing it as an argument the CME at 17¦1550.

write_page checks this page for zeros. It does not contain zeros.

write_page calls allocate_pd to see if this page requires allocation to the paging device.

allocate_pd notices that this page is already on the paging device (at record 101), and returns this fact to write_page.

write_page threads the CME at 17¦1550 out of the used list, marks the PTW (at 17¦21532) out-of-service, and marks the CME out-of-service on a write.

write_page calls the bulk store DIM to write the main memory frame at location 264000 (described by the CME at 17¦1550) to record 101 of the bulk store.

write_page returns to claim_mod_core.

claim_mod_core inspects the next CME that was in the used list, at 17¦1304. This CME was skipped because its PTW at 17¦16120 described a recently used page. The bit indicating this (ptw.phu) in this PTW is turned off, as demanded by the main memory replacement algorithm.

claim_mod_core inspects the next CME that was in the used list, the one at 17¦2200 was skipped because it designated a page which required migration to the paging device.

claim_mod_core calls write_page, passing the CME at 17¦2200 as an argument.

write_page notices that this page is pure, and does not check for zeros.

write_page calls allocate_pd to see if this page needs allocation to the paging device.

allocate_pd sees that the bit ptw.nypd is on in the PTW at 17¦14140, and determines that this page must be allocated a record of paging device.

allocate_pd inspects the first PDME on the PD used list. It is at 17¦6044, describing record 11 of the bulk store, and is free.

allocate_pd moves the PDME at 17¦6044 to the tail of the PD used list, and fills it with information from the CME at 17¦2200. The CME at 17¦2200 is changed to designate record 11 of the paging device as the home for the page.

allocate_pd returns to write_page the fact that a PD record was allocated during this call.

write_page sets the CME at 17¦2200 out-of-service, threading it out of the used list, and marks the PTW at 17¦14140 out-of-service.

write_page calls the bulk store DIM to write the main memory frame at 240000 (described by the CME at 17¦2200) to record 11 of the bulk store.

write_page returns to claim_mod_core.

claim_mod_core notices that sst.usedp and sst.wusedp are equal, and all operations skipped by find core have been processed.

claim_mod_core returns to the page fault handler.

The page fault handler determines that the PTW for the page faulted on (page 0 of segment 244) is still marked out-of-service (at 17¦24660). Since it is not a page on the paging device, the traffic controller will be used for waiting.

The page fault handler meters the page fault and the time spent in processing it.

The page fault handler develops the event ID for the page faulted on, 024660, and stores it in pds$arg_1 for pxss. The bit cme.notify_requested is set in the CME at 17¦2214.

The page fault handler branches to pxss$page_wait, with the page table locked.

pxss$page_wait locks the traffic controller lock.

pxss$page_wait unlocks the page table lock.

pxss$page_wait sets the process to waiting on the event 024660, and uses the PRDS frame to switch processes to another process.

Our process goes waiting.

The I/O operation in location 700000 completes, and the disk DIM interrupt side calls page$done with this number as a parameter.

page$done locks the page table lock and calls page_fault$done_ with that main memory address.

page_fault$done_ locates the core map entry at 17¦1340, and seeing that an RWS was in progress there, calls the routine read_write_sequence in done_. The PDME at 17¦6204 is located.

page_fault$done_ notices that a write cycle completed. The disk address 401512 (nulled) is resurrected to 001512, and taken from the PDME at 17¦6204 and placed in the PTW at 17¦15263, which had contained the paging device address 000041.

page_fault$done_ frees the PDME at 17¦6204, and frees the main memory at location 700000, placing the CME at 17¦1340 into the core used list.

page_fault$done_ returns to page$done, which unlocks the page table lock.

page$done returns to the disk DIM.

The I/O in location 230000 completes. The disk DIM interrupt side calls page$done.

page$done locks the page table lock and calls page_fault$done_.

page_fault$done_ finds the CME at 17¦2214, and sees that no RWS was going on there.

page_fault$done_ marks the CME at 17¦2214 as no longer out-of-service, threading it back into the used list.

page_fault$done_ locates the PTW at 17¦24660 from the CME at 17¦2214. The directed-fault bit is turned on, allowing access to the page, and the out-of-service bit turned off.

page_fault$done_ notices the bit cme.notify_requested in the CME at 17¦2214, and calls pxss$page_notify with the event 024660.

pxss$page_notify locks the traffic controller lock, and notifies our process, which was waiting on event 024660, and sends a pre-empt connect to CPU B.

pxss$notify unlocks the traffic controller lock, and returns to page_fault$done_.

page_fault$done_ returns to page_done, which unlocks the page table lock.

page$done returns to the disk DIM.

CPU B takes a pre-empt connect, and calls pxss$pre_empt.

pxss$pre_empt locks the traffic controller lock, and performs a "getwork" operation, abandoning the process that took the connect.

pxss (getwork) finds our process ready, and switches to it, setting it as running.

pxss (getwork) returns to pxss$page_wait in our process.

pxss$page_wait unlocks the traffic controller lock, abandons its PRDS stack frame, and transfers to page_fault$wait_return.

page_fault$wait_return restarts the machine conditions (at pds$page_fault_data) indicating an Appending Unit address preparation fault.

CPU B's Appending Unit successfully fetches and uses the PTW at 17¦24660, and resolves the virtual address 244¦14 to absolute address 230014.

SECTION XVII

GLOSSARY

abort
: See RWS abort.

abs-seg
: Not a segment at all. A segment number used for addressing as a segment an arbitrary main memory, disk, or paging device extent, the location of the extent being a parameter at the time that it must be addressed. See PTW-level abs-seg and SDW-level abs-seg.

abs-usable
: A main memory frame, part of the paging pool, in a system controller that cannot be deconfigured. Only abs-usable page frames can contain abs-wired pages.

abs-wire
: Of a page or segment. To make that page or segment abs-wired.

abs-wired

1. Of a page. A page in main memory, which not only is wired, but may not be moved around main memory. Pages wired but not abs-wired may be moved around by abs-wiring pages or deconfiguring memory. Used principally for I/O buffers.

2. Of a segment. A segment having some or all of its pages abs-wired.

accept
: Of a physical volume. To make those supervisor calls, which, by placing label information in the PVTE for a given drive, establish the binding, or association, between that drive and that volume.

access control segment (ACS)
: A segment whose ACL effectively determines access to a resource. ACSs for peripheral devices are in >sc1>rcp, and are maintained and used by RCP. ACSs for logical volumes are usually, but need not be, in >lv.

activate
: To make a segment active. Done by getting an ASTE, reading the VTOCE of the segment, filling in the ASTE, and hashing it into the AST hash table. See active. The parent directory of a segment must be locked in order to activate it.

activation attributes
  or activation information
: Those attributes of a segment that are read from the VTOCE every time a segment is activated, copied into the ASTE, changed while the segment is active, and updated back to the VTOCE. Examples: current length, maximum length, date-time modified. See permanent information.

active

1. Of a segment. Having a page table (and AST entry) in main memory; the criterion for whether or not a segment is active is whether or not it is hashed into the AST hash table.

2. (loosely) of a page. Belonging to an active segment.

3. Of the paging device, or an instance thereof. In use, having pages being allocated, read and written from it. The bit fsdct.pd_active tells whether or not the paging device is active. See unflushed.

add_type

A subfile of page control device addresses (devadds) that specifies whether it is a record of disk, a record of PD, a main memory frame, or a null address.

append

(verb) To combine an address (effective address) produced by the processor control unit, with a segment number (the effective segment number) maintained by the appending unit, and produce, by fetching and inspecting PTWs and SDWs, either a main memory address or a page or segment fault.

appending unit (APU)

That portion of the 68/80 processor responsible for the implementation of segmentation and paging. It performs appending, maintains all segment numbers, performs control operations on its data, and coordinates the taking of faults.

appending unit cycle

One of the operations of the appending unit that results in an address being presented to the processor port logic. For an append that does not result in a fault, the last such address (final address) is the address of the data requested by the control unit. Other APU cycles are to obtain and modify PTWs or SDWs. Each CU cycle (see control unit cycle) may produce many APU cycles.

associative memory

A content-addressable semiconductor memory in the processor appending unit. There are two: the PTW associative memory, which maintains the last 16 PTWs fetched from main memory, and similarly the SDW associative memory. The associative memories can be cleared via the CAMS and CAMP instructions. Used for speed, to avoid continual PTW and SDW fetching from main memory.

AST

For active segment table. The collection of ASTEs that describe all of the active segments in the system. (See ASTE.) The AST is the uppermost part of the SST.

ASTE

For active segment table entry.

1. (ASTE proper). A collection of attributes, other than file map or page tables, describing an active segment.

2. That collection of attributes, taken along with the page table and file map of a segment.

AST hash table

A table, kept in active_sup_linkage, that holds the heads of hash threads, so that the UID of any segment may be used to find its ASTE, if it is active, or the fact that it is not active.

AST pool

There are four sizes of AST entries, those containing page tables of 4, 16, 64, and 256 PTWs. Those of each size form four <u>pools</u>, that are managed separately.

AST trickle

A mechanism implemented in the AST replacement algorithm that periodically updates, from the AST, the VTOCEs of segments whose VTOCEs would benefit by so being updated. It is driven by AST traffic.

AST used lists

One of seven lists of AST entries, the four normal lists being "used lists" of each size, and others being lists of ASTEs selected by some special criteria. See used list.

atomic

See unitary.

attached

1. A private logical volume is attached to a process if an entry in that process' KST so attests. Only private logical volumes that are attached to a given process can be used by that process.

2. There are definitions of this term relative to user-ring I/O and resource control.

auxiliary service

A service provided by a subsystem that, although it is not one of the fundamental ones for which the subsystem exists, involves much of the central code of the subsystem. See peripheral service and basic service.

bad track list

A reserved area of the volume header of a physical volume, that will be used to contain bad track information in future releases.

basic service

A service provided by a subsystem that is one of the fundamental ones for which it exists. See auxiliary service and peripheral service.

bit map

A one-bit-per-record map of all of the disk records usable for paging on a given physical volume. All bit maps (those for mounted physical volumes) reside in the FSDCT.

bootload

1. (verb) To initiate the operation of the Multics system, when it is down, i.e., to bring it up via issuing the BOS BOOT command.

2. (noun) The act of bootloading.

3. The life-span of a Multics hierarchy from time of bootload to shutdown, or next bootload, in the case where shutdown cannot be performed.

branch

As used in this document, a data structure in a directory that describes a segment or directory. A segment's branch contains a physical volume ID and VTOC index for the VTOCE of a segment or directory. The ACL, names, author, bit count, etc., of a segment may be found in or from its branch.

bulk store

A core memory storage medium used as a paging device. The term is used when it is not relevant that it is being used as a paging device as opposed to any other storage medium.

cache

A 2048-word semiconductor buffer memory in the processor port logic. An attempt is made to maintain the last 2048 words fetched from main memory in the cache. The cache provides a substantially faster access time than that of main memory. As each processor contains its own cache, strategies are needed to prevent confusion about main memory contents. See encacheable.

call side

Those programs in page control that are explicitly invoked by processes that need services performed upon segments. See fault side and interrupt side.

call-side wait coordinator

The program (page$pwait) used by call-side page control to wait for a given event, via an appropriate mechanism, and set whatever bits will be necessary to cause the occurrence of the event to perform the necessary notification.

claim

When the PD or main memory replacement algorithm selects a page frame to have its contents evicted, and thus be freed, the page frame is said to have been claimed.

connected

Of a process and a segment. A segment is said to be connected to a process, or vice versa, if the descriptor segment of that process contains an SDW that describes that segment, and is not faulted. See trailer.

connection failure
or forward connection failure

A situation that exists when a directory branch describes a certain VTOCE, but that VTOCE describes some other (or no) segment. This can be determined by comparing UIDs. This situation can come about by accident or by deliberate salvager action. See also reverse connection failure.

contract

The action performed by a program, the conditions that must be true when it is invoked, and the circumstances describing the meaning and validity of the result.

control unit (CU)

That portion of the 68/80 processor that is responsible for decoding instructions, performing indirections, and routing data around the processor. The control unit develops effective addresses of words in segments; see appending unit. The control unit status can be stored via the SCU instruction, when a control unit cycle is aborted due to a page or segment fault.

control unit (CU) cycle

A control unit operation resulting in an effective address being presented to the appending unit or port logic. Typical CU cycles are "instruction pair fetch," "operand fetch," "indirect word fetch," etc. Any CU cycle can result in a page or segment fault when appending is performed for that cycle. Restart of that fault retries the aborted CU cycle.

core

An obsolete term used in many program listings and comments for main memory.

core map

A page control data base in the SST that contains a four-word entry (CME) for frame of main memory. It is protected by the page table lock. See Section VI for its layout. See core.

core used list
or main memory used list

A used list of core map entries (CMEs) describing the order of recency of use of main memory frames.

crawlout

A cross-ring signal that causes abandonment of a stack in the inner ring. Crawlouts out of ring 0 require supervisor data bases to be cleaned up. Crawlouts result from hardware problems or faulty software, or damaged directories, and cause the software running at the time to be interrupted and not continued.

critical process pages
The first page of a process' descriptor segment and PDS. These two pages must be wired (the process is then loaded) before the process can actually run.

deactivate
Of a segment. To remove it from the state of being active. To deactivate a segment, its pages are driven out of main memory and paging device, its VTOCE updated from its AST entry, and the ASTE freed. See active.

deadlock
or deadly embrace
A situation wherein a process having a given resource is waiting for some other process to free a second resource, but, unfortunately, the process having the second resource is waiting on the first process to free that first resource. See locking hierarchy.

deciduous
Of a segment. A segment read in as part of the bootload tape in collection 1 or 2 (i.e., part of initialization's, therefore the initializer's, hardcore address space) and placed into the hierarchy by the program init_branches. Deciduous segments reside entirely in the hardcore partition. See also reverse-deciduous.

Examples: >sl1>pl1_operators_, >pdd>!zzzzzzbBBBBBBB>pds

defined
Of a logical volume. Either a mounted public logical volume, or a mounted private logical volume attached to a given process. A process is said to have a given logical volume either defined or not defined.

demount
1. Of a physical volume. To dissociate a physical volume from the drive on which it is mounted, stopping use of it by processes.

2. Of a logical volume. To remove the logical volume table entry for a logical volume.

demounter
The procedure demount_pv, that coordinates the demounting of physical volumes.

deposit
To deposit an address to the free pool of records of a given physical volume (bit map) is to mark it as free, and available for subsequent withdrawing. See withdraw and bit map.

descriptor segment (sometimes DSEG)
An array of hardware control words (SDWs) that specifies the mapping between segment numbers and either segments or taking a segment fault. Each process has its own descriptor segment; it is a segment, and may be paged, in which case it is described by the descriptor segment page table. The first page of the descriptor segment of a loaded process is wired.

desperation
Action taken by paging device allocator when there are no free PD records. Desperation consists of evicting some nonmodified page from the PD, if it can be found near the head of the PD used list.

devadd (device address)
A page control format for all main memory, paging device, and disk addresses. The upper 18 bits are a record number or address, and the lower four bits (add_type) specify whether it is main memory, disk, paging device, or null.

DIM
For device interface module. The program that contains the code for managing the physical operation (as opposed to logical use) of a device (viz., the disks or bulk store).

double-write

A disk write performed as a reliability feature after a successful paging device write. Double writing is controlled by a parameter on the DEBG CONFIG card, and tends to keep paging device pages pure.

eligible

A process is made eligible by the traffic controller at the time that the latter describes that the former should be allowed to consume main memory resources (i.e., take page faults). Only eligible processes can run, although they must be loaded first.

emergency shutdown (ESD)

A set of procedures, invoked via the BOS ESD command, that attempt to produce an orderly shutdown of the system after a crash has occurred. This shutdown must be performed in order to update the disk records and VTOCEs for segments that were active at the time of the crash.

encacheable

Said of a segment. A segment is encacheable if words of that segment are allowed to be put (and hence subsequently found) in a processor's cache. An SDW bit (sdw.cache) controls encacheability. This is used to control sharing and prevent confusion about cache contents. Segments accessible to the IOM or FNP are routinely nonencacheable.

entry

(loosely) same as branch.

evict

Of a page. To drive a page out of a given main memory frame, or PD record, by writing it, moving it, or simply changing the state of its data bases as appropriate. Note that the eviction of pages that are identical to copies on disk or PD does not involve writing.

exposed

A physical volume is exposed to an instance of a paging device if that instance is active while that volume is mounted.

fatal

Of a crash. A crash for which a successful emergency shutdown could not be attained. Fatal crashes involve salvaging all physical volumes mounted at the time of the crash.

fault side

Those programs in page control that are invoked in response to a page fault. See call side and interrupt side.

fault vector

A pair of instructions at a fixed location in main memory associated with a specific type of fault condition. When the processor recognizes such a condition, it "takes" the fault by executing these instructions. In Multics, they are always SCU (store control unit) and TRA (unconditional transfer).

file map

A mapping between the pages of a segment and disk record addresses on some physical volume. Each page is mapped into either one such address, or a null address, indicating zero contents. File maps appear in VTOCEs. When a segment is active, the file map is distributed between the various page control data bases.

file-system time

A 36-bit representation of real clock time used in directories and in recording date-time used, date-time-modified, and other storage system times. It is the upper 36 bits of a 52-bit clock time.

frame

or "main memory frame"

A 1024-word block (on a 1024-word boundary) of main memory. See page.

FSDCT

For file system device configuration table. A paged data base containing many global volume management parameters, and all bit maps.

fsmap

Either the bit map of the volume map of a physical volume, or the bit map in the FSDCT for that volume, when it is accepted.

fsmap tail

in the fsamp of a paging region that is not a multiple of 32_{10} pages, the bits of the last word of the fsmap that are not part of the valid portion of the bit map. They must be zero.

function

A body of code that performs some particular action as part of the operation of some subsystem. A function is used in this book to mean an important internal interface. See service for comparison.

half_lock
or read lock

A data object that allows many processes to perform certain actions upon a data object or data base, but does not let certain other actions begin until no processes are performing any actions at all (either first or second kind). A process performing the second kind of action causes any process attempting to do the first kind to wait. Also called "multiple-reader one-writer" lock.

hardcore partition

A partition used for holding the pages of the supervisor. It is always on the root physical volume (RPV).

hardcore segment

A segment addressable in the hardcore address space of all processes. All such segments are created via initialization.

hierarchy

A set of directories, segments, and volumes that describe each other completely. Normally each site maintains one hierarchy, although some maintain others for development use. All unique IDs and PVT indices, paging device information, etc., are valid only with respect to one hierarchy.

higher

Of locks. See locking hierarchy.

"hot" buffer

A vtoce-part buffer, that although not out-of-service, is known to have contents differing from disk. Hot vtoce-part buffers arise only as a result of write errors, and must be flushed at demount time.

inhibit

Of a physical volume. To prevent segment (and VTOCE) creation upon that volume, by the setting of a bit (pvte.vacating) in its PVTE. This can be done via the inhibit_pv command, or the sweep_pv command.

initialization

The set of programs that run when Multics is bootloaded, until it is up to command level. Initialization is responsible for creating the supervisor data bases, and building the hardcore address space of all processes, among other tasks.

instance

Of the paging device. The paging device, and all of the pages that are and have been on it, and its map, from the time its map is initialized to the time it is shut down or the last record is flushed from it, whichever happens first.

interrupt side

Those programs in page control that are called by storage system DIMs to respond to the completion of an I/O operation. The interrupt side is not only invoked on behalf of interrupts; "running" and other activity can invoke it as well.

kernel

Of page control. The ALM programs in the main path of the page fault handler.

known segment table (KST)

A per-process table describing the mapping between segment numbers in that process and storage system segments. The segments are identified via pointers to their branches (using other segment numbers in that process) and unique IDs. The KST also contains a list of private logical volumes attached to the process. The KST is a reverse-deciduous segment.

label
or volume label

The first Multics record of a physical volume. It identifies the volume, and gives parameters about its last use.

live

Said of a disk address. A disk address that represents a given record of disk and the data in it. See nulled for comparison.

loaded

A process is loaded when its two critical process pages have been wired. Processes are loaded by the traffic controller when they are made eligible. Only loaded processes can actually run.

lock

A data object used to serialize processes performing certain actions and using or modifying certain data bases. A process locks a lock before performing these actions or using these data bases, and unlocks it when done. Only one process may have a lock locked at one time. A process trying to lock a lock that is locked by (or *to*) another process must wait for that lock. Processes are also said to *hold* locks when they have them locked.

In Multics, locks are single words of storage that are zero when not locked and contain the process ID of the process that has it locked when it is locked. See protect.

locking hierarchy

A conceptual partial-ordering of a set of locks via the arbitrary relation "higher" (>). If lock A >lock B, and lock B >lock C, then lock A >lock C. There is no inverse, and two locks may be totally unrelated. The locking hierarchy is used to prevent deadlock. The rule used by the Multics supervisor, states that no process may wait for the unlocking of a lock unless that lock is higher than every lock it has locked (sometimes called the "Bensoussan Algorithm").

logical volume

A set of physical volumes defined as a group, to which the user may direct the creation of segments. The choice of physical volumes with a logical volume for a creation is a dynamic choice of the system, and segments may move automatically between physical volumes of a logical volume.

logical volume table (LVT)

A system data base (in the segment "lvt") that contains per-logical volume parameters for each mounted logical volume, as needed by the hardcore. Included in such information is access class, and the heads of a thread of PVTEs of physical volumes belonging to that logical volume.

LVTE

For logical volume table entry. See logical volume table (LVT).

machine conditions

A 40-word description of a processor state at the time of a fault or interrupt. It contains the contents of all program-accessible registers, the state of an aborted control unit cycle, and various other information. To restart a set of machine conditions is to cause the processor to load its registers from that machine state, and resume the interrupted program.

main memory
(formerly core)

The core or MOS memory device from which the processor normally fetches instructions and data. All pages must be in main memory to be directly used by the processor. See also core.

master directory

A directory whose quota is not derived from its parent, and cannot be returned to it. Master directories are the only directories whose sons logical volume can be different from that of the parent directory. The setting of quota accounts on master directories, and the creation and deletion of master directories, is controlled by master directory control in ring 1.

migrate

To migrate a page to the paging device is to allocate a PD record for it, and write it to that record. From then on, it will be read from that record, until it is migrated off it.

mounted

1. Of a physical volume. Being physically mounted, and having the PVT entry for the drive on which it is mounted filled with parameters of that pack (pvte.used will be on).

2. Of a logical volume. Having all of its physical volumes mounted (1), and having an entry in the LVT.

multiplex wait protocol

The technique used by call-side page control to wait for a large number of events in parallel; it involves the simplex wait protocol and waiting for an arbitrary event.

multistep operation

An operation consisting of many unitary operations. See unitary operation.

nondeactivateable activation

Same as semipermanent activation.

nondeciduous hardcore segment

A paged hardcore segment that is not deciduous. Such segments are not in the storage system hierarchy, and thus have no pathnames.

Examples: bound_file_system, bound_system_faults.

not-yet-on-paging-device (nypd)

A pure page in main memory that has not yet been migrated to the paging device. Such pages are important because their eviction requires migrating them to the paging device.

null

An address that represents a record of any device, and a logical page content of zeros. A null address in a VTOCE is represented by having its high-order bit on. In page control, it is represented by an "add_type" of zero. The low-order 17 bits of a null address contain a debugging code that reflects the manner in which it was generated.

nulled

Of a disk address. One that represents a given record of disk, but a logical content of zeros. Nulled addresses appear only in page control data bases, never in VTOCEs. They may not be reported to VTOCE file maps. See null and live.

oopv

For out of physical volume. A condition where no more free records exist on a physical volume.

orphan

(Of a segment or its VTOCE.) A segment that has a VTOCE but no branch in the storage system hierarchy. Orphans may result via certain actions of the salvager or certain crashes. They can be located via the sweep_pv command. See reverse connection failure.

out of service

Undergoing I/O. Does not imply inaccessible.

pack

A demountable unit of disk storage. Same as physical volume.

page

A 1024-word extent of data at a 1024-word boundary of some segment. Pages belong to segments; they can exist in main memory frames, or on disk records or PD records or any combination of those.

page fault

An exception condition detected by the processor hardware (the appending unit) when an attempt is made to use a PTW that specifies that some page of some segment is not in main memory. This is indicated by the bit ptw.df being off. This causes the unconditional execution of a specific fault vector that effects a transfer to the page fault handler.

paging region

That extent of a physical volume described by the volume map, in which all records described in VTOCEs reside.

page table

The array of PTWs that specifies the mapping between addresses in a segment and either main memory frames or page faults. The page table of a segment is part of the ASTE; only active segments have page tables. The SDW of a segment (a paged segment) contains the absolute address of its page table.

paging device (PD)

An optional storage device from which pages are read and written from main memory, on which copies of disk pages are maintained for faster access. Only a bulk store subsystem can currently be used as a paging device. See bulk store.

paging device map
or PD map
or PDMAP

A page control data base in the SST that contains a four-word entry (PDME) for each record of paging device. It is protected by the page table lock. See Section VI for its layout.

parasite

A segment residing on a physical volume that has no VTOCE (e.g., descriptor segments on the RPV). All such segments are currently on the RPV. Their existence implies the need for a "short RPVS" in the case of a crash.

partition

A region (extent) of a physical volume, other than the VTOC and label area, used for some other purpose than pages of storage system segments.

PD

See paging device.

PD flush

The software that runs in subsequent bootloads after a fatal crash which repatriates pages on an unflushed paging device to their disk records.

PDS

For process data segment. A per-process (reverse-deciduous) hardcore segment that contains all per-process information needed by a process other than that describing its segment number to segment mapping. The first page of the PDS of a loaded process is wired. See KST.

PD used list

A used list of paging device map entries (PDMEs), describing the order of recency of use of paging device records.

peripheral service

A service of a subsystem highly removed from the main path and procedures, that may only call very high-level interfaces of that subsystem. See basic service and auxiliary service.

permanent attributes
or permanent information

Those attributes of a segment, stored in the third vtoce-part of its VTOCE, that are rarely read or changed. Examples: UID pathname, date-time VTOCE created. See activation information.

perm wired

1. Permanently (i.e., since bootload) wired. See wired.

2. Sometimes used to mean unpaged, since such segments, indeed, cannot be removed from main memory in any way. See "temp wired."

physical volume

Same as pack. A (usually) detachable storage medium of disk storage, containing entire segments, and a VTOC containing VTOCEs describing these segments. Each segment resides wholly on one physical volume. Each physical volume belongs to one and only one logical volume.

physical volume table (PVT)

A wired table of entries (PVTEs) describing almost all of the per-pack parameters for mounted packs. There is information here for page control, segment control, and volume management. The bit map for the pack, and per-logical-volume information is not stored here.

port logic

That portion of the 68/80 processor that selects system controllers, transfers commands and addresses to them, and receives data and notification from them. The port logic receives data from the processor, but addresses only from the appending unit. It contains the cache as well.

post

To post an I/O operation that was initiated by page control is to perform those actions taken by the interrupt side when told of the completion of this action by the appropriate DIM. Posting operations may or may not involve notification.

post-crash PD flush

See PD flush.

private

A logical volume is private if it was registered with this attribute. A private logical volume must be explicitly attached by any process that wishes to use segments on it; this is done conditionally depending upon the ACS of that volume.

preacceptance

The actions taken in initialization to use the partitions, including the hardcore partition, on the RPV, before the RPV has been accepted. Preacceptance of the RPV is performed by init_pvt.

prewithdraw

To assign a disk record address to a page of a segment at the time the segment is created, or at a given explicit time, as opposed to time of first use. All supervisor segments have their pages prewithdrawn by make_sdw.

preseek

An action taken by the main memory replacement algorithm to find a page frame to claim. It "looks ahead" for a usable page, postponing writing (see "writebehind") for later.

private

Of a logical volume. One to which access by users is restricted to those specified by the ACL of the ACS for that volume. Users wishing to use segments on private logical volumes must explicitly attach them. See public.

protect

A lock protects a data base or data objects, and/or operations on it, if such operations on that object or data base cannot be undertaken unless the process attempting to do so has the lock locked. For instance, the AST lock protects deactivations.

pseudoclock

A counter that is incremented every time an event occurs. Via appropriate protocols, an old value of a pseudoclock may be saved, and compared with a new value, an equal comparison implying that no occurrences of the event have happened.

PTW

For page table word. A processor hardware control word, an element of a page table, that specifies either a main memory frame address or that the processor should take a page fault when attempting to use this PTW.

PTW-level abs-seg

An abs-seg implemented via a page table; the SDW for this segment number describes that page table, and the PTW contents and PVT index in the ASTE are varied to describe the extent of disk or bulk store being addressed.

public

Of a logical volume. One on which access to segments is restricted solely by the ACLs on the segments. See private.

pure
or purify

A pure page is one that has a good (i.e., identical) copy on secondary storage or the paging device. To purify a page is to write it out so that this is so.

PV hold table
or PVT hold table

A table of half-locks, protecting nonunitary VTOC operations against the physical volume demounter. Also used to schedule salvages if crawlouts occurred with a volume "held" (half-locked).

PVT

See physical volume table.

PVTE

For physical volume table entry. See physical volume table.

quota

An administrative limit on disk record consumption. The quota of a directory is the maximum number of nonzero or in-main-memory pages allowed to be created for segments charged to the quota account of that directory.

quota account
A data structure associated with a directory (in the VTOCE and/or ASTE of that directory) that allows segments inferior (not necessarily immediately inferior) to that directory to have their record consumption charged against a single pool. See quota.

quota cell
The information in an ASTE or VTOCE for a directory with a quota account that describes the quota limits and records currently charged against that quota account.

read-write sequence (RWS)
A sequence of operations by means of which the copy of a page on the paging device is written back to a record of disk. This is only performed for pages for which the paging device copy is different than the disk copy. An RWS consists of allocation of a main memory frame, reading in the page from bulk store, and writing it to disk, freeing the frame when done.

record
1. (Disk record) A 1024-word, contiguous extent of disk that can hold a copy of a page.

2. (PD record) A 1024-word, contiguous extent of paging device that can hold a copy of a page.

repatriation
The act of locating the segment to which pages "trapped" on an unflushed paging device belong, and writing these pages back to the appropriate disk record. Performed by the post-crash PD flush.

residue
The data left over in a record of disk or bulk store or a frame of main memory after a given page no longer resides there. It is impossible to read residues.

resource control package (RCP)
Multics subsystem (running in ring 1) that controls and mediates access to peripheral devices. RCP also controls the attachment (see attached) of private logical volumes to user processes.

resurrection
The act of converting a nulled address into a live address, which allows it to be reported to a VTOCE file map. See nulled. Resurrection is performed upon the successful completion of a write, double write, or RWS.

reused address
A disk address simultaneously in use by two different pages. Such a situation is theoretically impossible. See unprotected address.

reverse connection failure
A situation that exists when a VTOCE describes a segment, but no branch in any directory in the storage system describes that VTOCE (or therefore that segment). Such a segment is said to be an orphan. See connection failure.

reverse-deciduous
A segment in the storage system hierarchy that is placed into the hardcore address space of some or all processes via semipermanent activation. Examples are the PDS of any process except the initializer and >online_salvager_output. See deciduous.

RLV
See root logical volume.

root logical volume (RLV)
The logical volume, which contains the RPV, on which all directories exist. It is the only logical volume necessary for system operation.

root physical volume (RPV)

The disk, residing on the drive pointed to by the root config card, on which the root directory (>) and the hardcore partition (in which the supervisor resides) exist. It is a member of the root logical volume.

RPV

See root physical volume.

RPV-only directory

A directory, whose sons logical volume is the RLV, whose inferior segments and directories may only be placed on the RPV. The root is such a directory, as is >lv.

RPVS

BOS keyword for root physical volume salvage. A volume salvage of the root physical volume (RPV), that is performed by the system during initialization when necessary or requested. Most cases of RPVS also involve some automatic directory salvaging. See short RPVS.

run

Said of disks, the bulk store, or their DIMs. To call the appropriate hardware interface modules for a device, and see if operations have completed, invoking the interrupt side of page control when this has happened. One can "run" a device in a loop until an arbitrary number of operations or an arbitrary operation has completed. Simulates an interrupt, in effect.

RWS

See read-write sequence.

RWS abort

The action taken by the page fault handler when a page fault is taken on a page from which an RWS is in progress. When the RWS is posted as complete, the page fault will be resolved by the interrupt side of page control.

scrap

Of paging device records on an unflushed paging device. To cause the system to ignore the contents of such records, forgetting the fact that they are unflushed and in need of repatriation.

SDW

For segment descriptor word. A hardware control word, an element of the descriptor segment, that gives the absolute address of an unpaged segment, or the absolute address of the page table of a paged one. The SDW also contains access mode and ring brackets, as well as other information. The SDW can also specify taking a "segment fault."

SDW-level abs-seg

An abs-seg implemented as a variable SDW slot; various SDWs either describing main memory or page tables are inserted in that slot to describe the main memory extent or segment to be addressed.

secondary storage

Permanent storage as opposed to the paging device. Interchangeable, in most contexts, with "disk."

segment fault

An exception condition detected by the processor (the appending unit), when an attempt is made to use an SDW that describes a segment not yet connected to the process in whose descriptor segment the SDW appears. This is indicated by the bit sdw.df being off. A segment fault causes a specific fault vector to be unconditionally executed, ultimately invoking the segment fault handler.

semipermanent activation

Activating a segment in such a way that it will remain active even after the AST is unlocked. This is performed by the program grab_aste, and is done by turning on the bit aste.ehs, the "entry hold switch."

service

An action performed by a subsystem on behalf of some other subsystem, a class of action so provided. The services performed by a subsystem are its reason for existence. See function for comparison.

setfaults

An operation performed by the procedure of the same name, at the time a segment is deactivated or its access attributes are changed. This operation modifies or faults all of the SDWs for a given segment, located via the trailer list. The associative memories of all processors are always cleared as the last step of a setfault.

short RPVS

A root physical volume salvage (see RPVS) performed automatically upon bootload after a successful emergency shutdown. It is called "short" because no directory salvaging of any kind is performed in this case.

simplex wait protocol

The technique used by the page-fault handler and the multiplex wait protocol to await the occurrence of a page control event. This technique involves the assumption that the "occurrence" of the event may not have happened when indicated, and the status of the operation being performed having to be reevaluated.

sons logical volume

Of a directory. The logical volume on which all segments created inferior to this directory will reside. A directory inherits its sons logical volume from its parent unless it is a master directory.

SST

For system segment table. A supervisor segment (sst_seg) that contains almost all page control data bases, all AST entries, and many meters. It is contiguous in main memory (unpaged), as it contains page tables used by the hardware. See core map, PD map and AST.

temp wired

1. Temporarily wired, via calls to the wiring interfaces (pc_wired) in page control.

2. Sometimes used to mean paged and wired, as opposed to unpaged. See perm wired.

trailer

An entry in the system trailer segment (str_seg) attesting to the fact that a process has an SDW for a given active segment. Each active segment possesses a <u>trailer list</u> of such processes. The trailer identifies the process via the AST offset of its descriptor segment's ASTE, and contains the segment number of the segment in that process. See setfaults.

trickle

See AST trickle.

unflushed

Said of the paging device, or an instance thereof. Containing pages from a previous bootload, that are in need of repatriation. See active for comparison. No new pages are migrated to an unflushed paging device, and the system will not come up if it has one.

unique ID (UID)

A 36-bit number assigned to a segment at the time it is created. It is different from any other UID for any other segment in that hierarchy. It is stored in the VTOCE, ASTE, KSTE, and branch for a segment, and must match for all of these objects. UIDs are also stored on the paging device to facilitate repatriation.

unitary
  or atomic operation

An operation performed upon a data base or data object by a process, in such a way that no other process attempting to perform or succeeding in performing the same or other operations upon that data base can affect the operation being performed in any way.

unprotected address

A disk address in use by some page of some segment that is not marked as in use in the bit map for that volume. Such a situation is theoretically impossible.

used list

See PD used list, AST used list, and core used list.

"Used lists" in Multics are circular, double-threaded lists of similar objects, containing both free and in-use objects. All of the free objects are maintained at the head of the list.

Used lists generally implement replacement algorithms, with the entries at the head of the list that are not free the most likely candidates for replacement.

vacate

1. Of a physical volume. To drive all of the segments on a physical volume onto some other physical volume in that logical volume. Done by the sweep_pv command.

2. Of a main memory frame or PD record. To evict any page from that frame or record, such that the frame or record becomes free.

volume header

The first eight records of a physical volume, containing the label, the volume map, the VTOC header, and the bad track list.

volume map

A data base in the volume header of a physical volume that describes the paging region of that volume, and includes a bit map telling which records are in use.

VTOC

For volume table of contents. An array of entries (VTOCEs) describing each segment on a physical volume. The VTOC occupies a fixed, contiguous extent at the beginning of a physical volume.

VTOCE

For VTOC entry. A disk-resident data object that describes one segment on the physical volume on which it appears; this description includes record addresses and other attributes.

VTOC header

A data base in the volume header of a physical volume, that tells the extent of the VTOC, and contains the head of the free VTOCE thread.

vtoce-parts

One of the three physical 64-word parts of a VTOCE. The first vtoce-part contains the activation information, the third the permanent information. The file map is in all three.

wire

Of a page or segment. To make that page or segment wired.

wired

1. Of a page. A page that may not be removed from main memory. The bit ptw.wired tells page control not to replace this page.

2. Of a segment. A segment having some or all of its pages wired. See also abs-wired.

withdraw

To withdraw an address from the free pool of records of a given physical volume (bit map) is to request an unused record and mark it as used, obtaining the address of that record. (Also said "address withdrawn against a given bit map.") See bit map and deposit.

writebehind

A feature of the main memory page replacement algorithm whereby the basic path of the algorithm skips (see preseek) writing, and this writing is done later (at the end of page fault processing).

APPENDIX A

CHANGES FOR MR 6.0

This appendix describes storage system implementation details that are markedly different in Multics Software Release 6.0 from descriptions found elsewhere in this manual. Areas affected, and described in this appendix, are:

1. prewithdrawing policy
2. per-process hardcore segment policy
3. volume dumper support
4. page posting queue
5. page control traffic control interface
6. page control consistency strategy
7. page control error strategy
8. large volume map space
9. damaged segments
10. quota validator
11. support of hierarchy salvager
12. limited update backlog
13. partial shutdown

PREWITHDRAWING POLICY

The algorithm given in Section IV under "PDS and KST Management" for prewithdrawing segments is incorrect. Step 4, if it causes a segment move, causes all the prewithdrawn addresses to be deposited during the segment move. Instead of the conjunction of bits aste.dnzp and aste.ehs indicating "don't deposit nulled pages" (5.0 policy), a new bit, aste.ddnp, indicates precisely this. This bit inhibits reporting of nulled addresses for deposition in pc$get_file_map and pc$list_deposited_add. Thus, segments with this bit on in the ASTE do not get nulled addresses reported to segment control, even after truncation. What is more, the segment mover copies this bit into the new ASTE of a segment move before copying the data. Since nulled pages have disk addresses at the time the segment mover attempts to copy the data, it copies, and thus prewithdraws, nulled pages against the new segment, exactly as desired.

The bit aste.ddnp is seen by the AST replacement algorithm as an entry-hold switch; it is metered against steps-ehs. Thus, no segment with this bit on can be deactivated. However, it can be segment moved, and its ddnp-quality preserved. Thus ddnp implies that the segment ought not to be deactivated because of the prewithdrawn quality of its addresses, as opposed to someone sequestering the page table address, which is the normal reason for setting an entry-hold switch.

When ddnp segments are released from the entry-hold state, the ddnp switch is turned off so that addresses of the segments may be reported and deposited after truncation.

Notice that ddnp does not imply dnzp or vice versa; segments with prewithdrawn addresses can benefit perfectly well from not having zero pages written out/read in, but just created in place.

The entries grab_aste$prewithdraw and grab_aste$release_prewithdraw are used to prewithdraw segments (turning on ddnp) and turn off ddnp. In fact, the existence of ddnp simplifies grab_aste radically. This routine now uses this bit (turning it on only to turn it off later, if not the $prewithdraw entry) to force a segment into a sufficiently large ASTE. The algorithm given in Section IV under "Semi-Permanent Activation" is no longer necessary. By turning on aste.ddnp, and touching the needed page, without storing into it, grab_aste can be sure that the segment cannot be deactivated as long as aste.ddnp is on. When a boundsfault occurs, boundfault moves the page's PTW as well as the bit aste.ddnp, causing the page not to go away until the bit is turned off.

PER-PROCESS HARDCORE SEGMENT POLICY

The descriptor segment of a process is now a reverse-deciduous segment. So are all PRDSs, except the bootload-time PRDS, which is deciduous. This has the effect of eliminating parasitic segments (see Section VII, "RPV Parasite Segments") for all cases except scratch segments used by the volume salvager and disk rebuilder. Thus, in release 6.0, descriptor segments appear in the hierarchy, in the process directory. They are prewithdrawn as are PDSs, as described in "PDS and KST Management" in Section IV.

The motivation for doing this was to eliminate the need for the "short RPVS" performed automatically after a successful ESD. The need for the short RPVS (see Section VII) was engendered by addresses withdrawn from a volume map but neither deposited nor reported to a VTOCE (for deletion by normal means) at shutdown time.

Thus, the pages occupied by descriptor segments at the time of a successful emergency shutdown are deposited when the old PDD is deleted by delete_old_pdds. This places descriptor segments on any packs where process directories go, as opposed to constraining them to the RPV. Although this has the effect of diluting the I/O load of the RPV, this subjects segment control (setfaults in particular) to the vagaries of many disks.

In release 6.0, therefore, there is no "short rpvs". The volume salvager and the disk rebuilder use the PV hold table (See Section XIV) to cause full volume salvaging of the RPV if the system should crash while their parasitic temporary segments are in use.

PRDSs for all configured CPUs are created and entry-held (and prewithdrawn) for all configured processors at bootload time, by tc_init.

The program "plm", which was used to create parasitic segments, no longer exists. Descriptor segment initialization logic was moved into act_proc, creator of processes.

VOLUME DUMPER SUPPORT

Unlike hierarchy backup, the new volume backup facility is an integral part of the supervisor. Volume backup accesses segments directly via their VTOCEs, avoiding the overhead of scanning directories to seek out and initiate these segments. There are several important facilities of the volume dumper in segment control, and several important ramifications of its existence.

The volume dumper maintains in the label area of each pack (see Section XIII in the "VTOC HEADER" records, #4 and 5), a bit map of segments that have been modified since dumped. When a volume is accepted, load_vol_map causes (via the program dbm_man) a region to be allocated in the global hardcore segment dbm_seg to contain this map. It is read in, and a pointer to it left in the PVTE for the volume. When the label of the volume is written back, so is the "dumper bit map".

When the volume is in use, all primitives that modify or detect modification to a segment or a VTOCE (notably pc$get_file_map, truncate_vtoce, create_vtoce, and delete_vtoce) call an entry in dbm_man to set the bit corresponding to that VTOC index (the dumper bit map is indexed by VTOC index).

Like record address depositing, setting dumper bit map bits is an operation that must be protected from volume demounting in the window after segment modification has been noted. Thus, the demount protection brackets (described in Section XIV under "Demount Protection") protect dumper bit map setting as well. Since this is now done in create_vtoce, this program now uses the demount protection-bracket mechanism where it did not before, and thus the unitary quality of vtoc_man$alloc_and_put_vtoce is no longer necessary.

The incremental volume dumper scans the dumper bit map to locate segments to be dumped, turning off the bit once they have been dumped. The volume dumper dumps segments and directories in the same way: it dumps binary images. (The volume dumper does, however, lock directories by UID--anonymously, i.e., no pointer--when dumping them, in order to get a consistent copy of the binary object, i.e., no one should be modifying it.)

The volume dumper (incremental, consolidated, or complete) accesses segments via a special entry to "activate", which activates a segment given its PVID and VTOC index, without its branch. This "parentless activation" is performed only for the volume dumper. When the volume dumper wishes to activate a segment for dumping, activate first hashes it into the AST (as for any activation) to see if it is already active, and returns the AST entry pointer if it is active. When activate so does, it turns on the "dumper in use switch" (aste.dius) to prevent any other process from deactivating the segment (get_aste knows to skip such segments). If the segment is not active, activate activates it again setting aste.dius. Any other attempt to activate this segment finds this ASTE, as it is hashed in normally. The bit aste.dius is not turned off until the dumper is finished. When parentless activation is accomplished for the volume dumper, quota checking is suppressed (aste.nqsw turned on) so that page control does not chase up a nonexistent parent pointer.

When the dumper finishes dumping a segment, it must deactivate it if it activated it, and the segment has not yet acquired a parent pointer.

The dumper uses the hardcore segment number of "backup_abs_seg" to construct abs-segs to reference the segments it activates by the above means. It puts itself on the trailer of the segment (see "Trailers and Setfaults" in Section II), such that if the segment is deleted while the dumper is dumping it, the dumper takes a fault, and cleans up. The various entries in trailer management are cognizant of the possibility of a trailer with a hardcore segment number for this purpose.

The volume retriever operates by using the standard VTOCE/segment creation primitive (create_vtoce) to create new segments, creating the VTOCE and segment for an extant branch if there is one (forward connection failure). If a VTOCE (and thus segment) already exists for a segment being retrieved, it simply copies the new data into it. Directory contents are merged (see the program retv_copy) in ring zero. When the volume retriever is called upon to retrieve a segment whose branch does not exist, a special entry in directory control's "append" primitive is called upon to create a branch from saved binary information, as opposed to user-supplied symbolic data, without regard to access checks. In this case, the VTOCE/segment being retrieved is connected to this branch.

The volume reloader is not part of the supervisor; it constructs complete physical volumes from volume backup tapes, placing VTOCEs and segment contents on it as appropriate. It uses user-ring disk I/O, and works on volumes not now in use by the storage system.

The segment adopter, and the -adopt option to sweep_pv construct a directory branch for a segment whose VTOCE is extant, but has no branch. As such, it is not part of segment control. The primitive used by both is the same entry to directory control's append primitive used by the volume reloader to construct a branch for an item whose VTOCE (and actual data) it is retrieving.

PAGE POSTING QUEUE

Page control posting strategy (see "I/O Posting" in Section VIII) has been modified to make it no longer necessary for the disk DIM interrupt side to loop-lock the global page table lock. On multi-cpu systems, where the disk DIM interrupt side (on a real interrupt, as opposed to a run) can find the page table lock locked, this loop locking consumed a substantial share of system resources prior to release 6.0.

The solution to this problem was to construct a queue of coreadd/errcode (the parameters to page control supplied by the disk DIM in a posting call) pairs that the disk DIM wanted to report to page control, but rather would not, because the page table lock was locked at the time. Any program at all that unlocks the page table lock is responsible for checking out this queue, and calling page$done_ (see Section VIII) with each coreadd/errcode pair in it.

This queue is called the "disk posting delay queue", or the "coreadd queue", due to its content, and resides in the segment "disk_post_queue_seg". The locking policies involved make sure that everything that is put in the delay queue is processed as soon as possible, and that no requests are lost are quite involved, and are further described below. The maintenance of the delay queue is all performed in the program core_queue_man, in bound_page_control.

The locking policy of the coreadd queue may be expressed as follows: The coreadd queue has a lock on it (in disk_post_queue_seg), which must be loop-locked. This lock is higher than the page table lock (see "Locking Hierarchy" in the glossary). Absolutely no lock-looping is performed with the coreadd queue locked (i.e., no lock is higher than the coreadd queue lock). The only code in the entire system (during normal operation, i.e., not ESD) which unlocks the page table lock is in core_queue_man, and does so only while the coreadd queue is locked. Thus, before actually unlocking the page table lock, a potential unlocker of the page table lock is in a position to inspect the coreadd queue. If the queue is empty, the page table lock can be unlocked and then the coreadd queue lock can be unlocked. If there are posting requests in the coreadd queue, the first one must be taken out, the coreadd queue unlocked (the page table lock is still locked), and call page$done_ while the page table lock is still locked, to perform the posting. It is then necessary to try again to unlock the page table lock, starting by locking the coreadd queue lock. Among the unlockers of the page table lock are page$done, called by the disk DIM interrupt side, in the case where it did not initially find the page table lock locked (i.e., could lock it), and has called page$done_. An attempt must be made to lock the page table lock before the coreadd queue is unlocked when a process has locked it to queue a request for posting there; otherwise, the request might stay forever in the coreadd queue if the page table lock indeed became unlocked between the time the interrupt side found it locked and the time the interrupt side locked the coreadd queue lock (once the coreadd queue lock is locked, the page table lock cannot be unlocked except by the process which has it locked). If the attempt to lock the page table lock is successful, all postings must be done by this process. If not, the process can rest assured that whoever has it locked is going to have to unlock it, and can't possibly unlock it until the current process unlocks the coreadd queue lock, which it has locked, and thus, that other process will find the request just queued as soon as it unlocks the page table lock.

The coreadd queue must be processed at "run" time, and flushed at ESD time as well.

See core_queue_man.alm for more information.

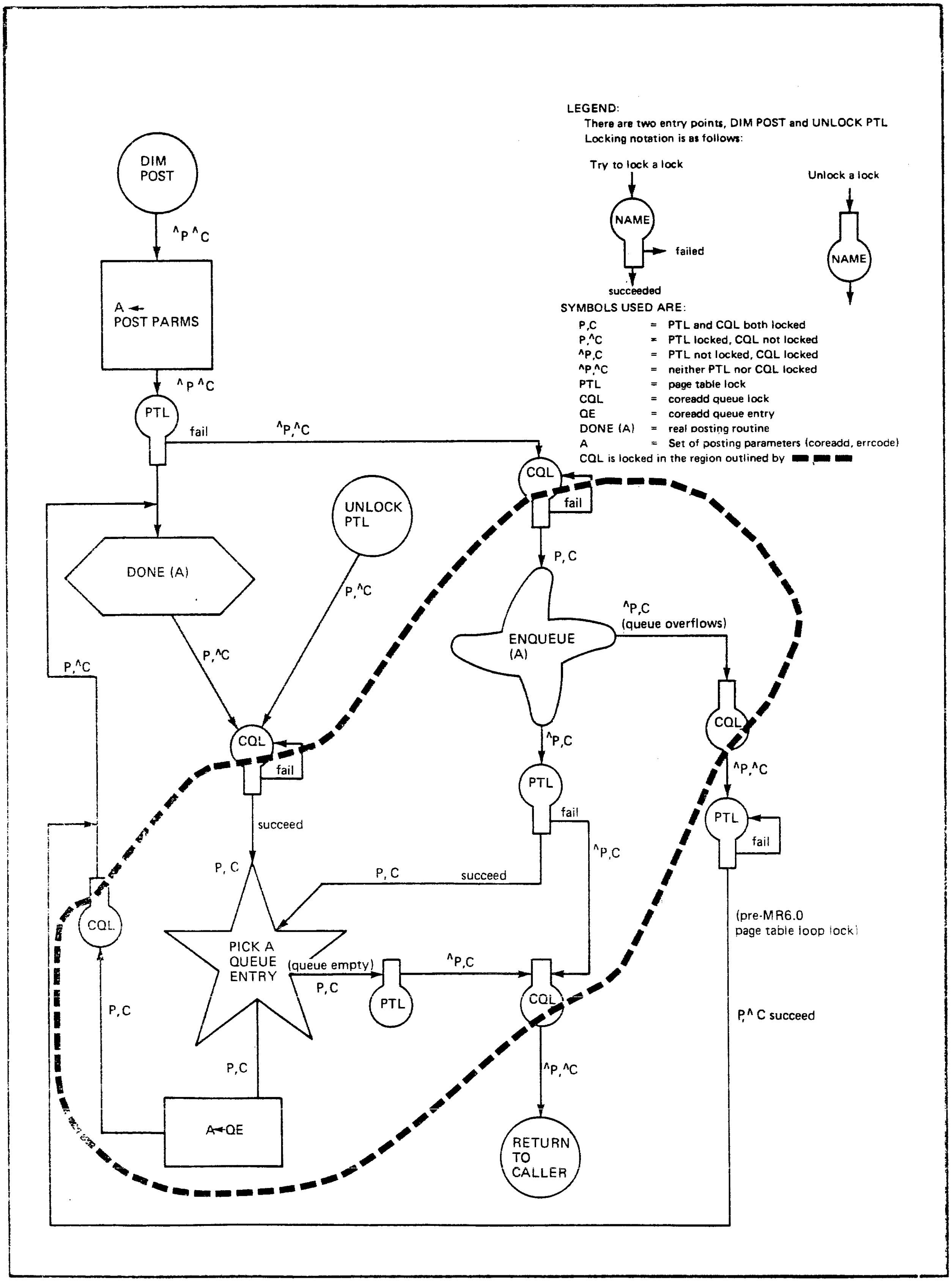


Figure A-1. Coreadd Queue Locking

PAGE CONTROL TRAFFIC CONTROL INTERFACE

The implementation of the disk posting queue involved cleanup in page table locking and unlocking. The unlocking of the page table lock under protection of the traffic controller lock (Section VIII under "Stack Management and Interface with the Traffic Controller") is no longer done. In release 6.0, page control unlocks the page table lock before the traffic controller lock is locked, when going to wait. Taking advantage of some features of the new lockless scheduler, page control does a standard "addevent" when it is going to branch to the traffic controller, storing a wait event (which it knows has not yet been notified, this decision made under the protection of the page table lock, under which all page control notifies are done) in the APT entry of the waiting process. If the traffic controller finds, under the traffic control lock, that this event has been notified (become zero), the traffic controller returns to page control to restart the fault or call side operation.

These changes allowed a new mechanism for waiting for the page table lock from the call side to be implemented. When the call side of page control attempts to lock the page table lock (in device_control.alm), a branch is taken to the traffic controller for page-table lock waiting if it cannot be locked. By the identity of the entry point called, as encoded in the value of pds$pc_call (as for waiting for paging events), the traffic controller returns to device_control$dvctl_retry_ptlwait to reattempt to lock the page table lock when it has become unlocked.

Thus, the only times that the page table lock is looped on are at process-loading time, and if the coreadd queue is full.

Page control no longer uses regular traffic-control waiting for the page table lock; a special traffic controller state is used exclusively for this type of waiting. Also, traffic control returns to a point in the page fault handler instead of restarting a page fault when ptlocking-waiting is complete, in order to avoid the fault overhead. This relies on the fact that the page fault data stored in the PDS cannot have possibly changed since the page fault was taken.

Traffic control no longer needs to validate page control events under the traffic control lock (described in Section VIII under "Global Page Lock"). The above interface wherein page control stores wait events directly in the APT entry of a waiting process (even during the process loading function) obviates the need for this validation. If an event becomes invalid, a notify clears it out of all APT entries.

PAGE CONTROL CONSISTENCY

Until release 6.0, emergency shutdown (ESD) has been a fairly risky proposition, because of the unknown state of page control data bases at the time the system crashed. Whether or not the system crashed in page control, or because of some problem detected in page control, there was not (and is not now) anything to prevent the system from crashing when some CPU was in page control or the disk or bulk store DIM. This is critical because ESD relies on the correct functioning of page control, not only to write out pages of segments, but to support the virtual memory in which much of ESD runs. The assumption that page control could be used reliably after a crash was therefore not always valid: inconsistently threaded data objects could often cause faults to be taken, and I/O requests in the process of being queued or posted often get lost, causing the system to hang indefinitely awaiting their completion. At worst, these inconsistencies led to misrouting of data, and most often to failure of emergency shutdown in one way or another, with all concomitant grief.

Thus, all data and state manipulation in page control was redesigned and reimplemented to make the following statements true at every point (at all times):

1. If page control is interrupted at this point, a procedure running at ESD time can compute distinctly, fully deterministically, a valid state of the entire data base of page control, reflecting its state either before or after a database change that was interrupted completed or would have completed.

2. If page control (or the disk or bulk store DIM) is interrupted at this point by a system crash, a procedure running at ESD time can regenerate any I/O that was queued, in progress, or in the process of being queued, posted, or performed, without fear of the original I/O ever being posted.

The "procedure running at ESD time" is pc_recover_sst in bound_page_control, also well worth time reading. This procedure places the entire page control data base (the SST) in a consistent state before any paging or page control operations are attempted by ESD.

The fundamental truth that allows this technique to operate is that very little of page control is actually changing the data base, or therefore, the state of page control. Most of page control is making decisions, and calling subroutines. It is only at the very lowest level, almost entirely in ALM page control that the data base is changed. Most of PL/I page control is simply making decisions and mapping the actions of ALM page control over segments. Thus, in order to recompute the consistent state of interrupted page control, we need not know what decisions were being made, or what segment-wide operations were being performed. All low-level page control operations involve only one page of one segment; when one page replaces another in memory, this is really two operations: an eviction and a paging-in. Between the two operations, the main memory frame is distinctly free. During the eviction, or during the paging in, the page under consideration is either in main memory or not: there is no inconsistency involving two pages. Other page control operations are comparably defined.

Typical of the operations under consideration that may be interrupted and must have their state recomputed are:

1. Binding a page of a segment to main memory (paging-in),
2. Unbinding them (eviction),
3. Binding a frame of PD to a page of a segment (PD Migration),
4. RWS initiation,
5. RWS completion, and
6. Unbinding a page from a PD record, either at RWS completion time or during PD Housekeeping.

Each of these operations involves the establishment or revocation of bindings between at most one page of one segment, one main memory frame, and one PD record. As a matter of fact, each such operation consists of the establishment or revocation of at most one (usually bilateral) binding. Each such bilateral binding is usually two values that designate each other. For instance, the binding between a page of a segment and a page of main memory is expressed by the fact that a PTW has a main-memory type address in it, designating a CME that has the address of the PTW in it. The binding between a page of a segment and a record of paging device (PD) is expressed by the page of the segment (PTW pointer) being in the PD map entry, and the PD address being in either the PTW or CME, depending on whether or not the page is in main memory. During a Read-Write Sequence (RWS), a similar bilateral binding between a PD record and a main memory frame exists in crossing pointers in the CME and PDME involved. Therefore, the establishment or revocation of any binding involves, in essence, the setup of two (perhaps conceptual) pointers. Bindings of objects are never changed (except in one case in evict_page, which is quite special) from "bound to this" to "bound to that", but only from "free" to "bound to this" or vice versa. Thus, every page control object can be viewed as "bound to something" or free at any instant, by looking at some critical pointer or field in it. For instance, if a CME has a nonzero cme.ptwp (or mcme.pdmep), it may be considered to be bound to that page of a segment (or PDMAP entry during an RWS), or else none. If a PTW has a main-memory type devadd in it, then that page is bound to that frame of main memory, or else none. The presence of a PD-type devadd in a PTW or CME (which itself is bound to some page (PTW)) says that that page is bound to that PD record, or else none. The presence of pdme.used in a PDMAP entry says that that PD record is bound to the page whose PTW is designated by pdme.ptwp, or else none.

Thus, certain critical fields determine distinctly, at any real time instant, whether or not a given object is bound to some other kind of object (and if so, which one). Before an object is marked a bound to some other object, all other fields except the critical field are filled in to their final values. If page control is interrupted before the critical field is filled in, pc_recover_sst finds the critical field not filled in (usually zero, see last paragraph), and the noncritical fields are essentially garbage; the binding is considered not to have started at all. If the critical field is found filled in, all other fields must be valid, and the binding was entirely complete.

The problem is therefore reduced to consistency between halves of a bilateral binding. This is accomplished by simply stating an order in which halves of bilateral binding are accomplished, the unbinding being accomplished in the opposite order. Thus, if pc_recover_sst finds two valid bindings, which are halves of a bilateral binding, the entire bilateral binding must be complete. If it finds one half of such a binding complete (after determining completeness by the rules of the last paragraph), it can either complete the binding or complete the unbinding, without regard to whether a binding or unbinding was in progress at the time page control was interrupted.

The following rules govern the establishment of bilateral bindings:

Pages to main memory frames, and vice versa:
when binding (reading-in), first bind the CME to the PTW, then change the PTW to designate main memory. When evicting, do the opposite.

Pages to PD records, and vice versa:
When binding (PD migrating--always happens when page in main memory), first bind the PDME to the PTW, and then change the CME to the PDME. When performing PD eviction, either at the completion of an RWS or a pure eviction during PD housekeep, do the opposite (i.e., first change the PTW or CME, then free the PDME). At all of these times (migration, RWS complete, pure eviction, and in-core PD eviction in pc.pl1) the copy of the page on disk or in main memory is, or is the same as, the most recent.

PD records to main memory frames, and vice versa (during RWS):
First bind the PDME to the CME, and then the CME to the PDME. At RWS complete time, do the opposite.

The handling of I/O in progress at the time of the crash is made trivial by the action of page$esd_reset, which calls entries in the disk DIM and the Bulk Store DIM to throw away the entire contents of their queues, and reinitialize their data bases. Thus, any page that is seen as out-of-service by pc_recover_sst may be simply evicted if it was a read in progress, knowing that the read is not actually in progress (the system crashed), and is not posted (the queues are flushed). If a write was going on, the modified bit is turned back on when this is done, because the action of initiating the write caused the modified bit to be turned off by write_page (the latter knowing that the page would be written). The modified bit is not turned back on, however, a page that is being updated as pure ("nypd write") to the paging device. The bit cme.pd_upflag, reclaimed for this purpose, indicates during a write that this is the case.

The routine pc_recover_sst can tell if the above rules have been violated, due either to bug, processor or memory malfunction, or damage to the page control data base by other parts of the operating system. Even in this case, it attempts to make the page control data bases consistent so that ESD can succeed. When such unexplained damage (i.e., inconsistency that cannot happen by virtue of the above rules) is detected, segments are marked as damaged and involved pages zeroed where appropriate.

The flushing of DIM queues at ESD time substantially simplifies the ESD strategy of the VTOC manager (see Section III, "ESD Strategy"). The VTOC manager can now decide distinctly that no I/Os queued before the crash are ever going to be posted. The bit b.ioq is now superfluous.

PAGE CONTROL ERROR POLICY

Release 6.0 makes radical changes to the handling of disk errors as detected by page control. First of all, errors are not reported to the operator console or the syserr log unless a page is actually damaged. The disk DIM has already reported all device error information for any I/O operation involved. A differentiation is made between device errors that affect a particular record gone bad, and those that are an indication of a device problem. In the latter case, it is almost always true that the operator can re-ready the device, or it will re-ready itself, or some nonautomatic remedial action can be taken. Thus, in any of these cases, it is unwise to perform irreversible action such as damaging a segment, or even wasting syserr log space with messages. The disk DIM differentiates between the device error case and other cases in the value of the error code at posting time. Errors reading either therefore replace the disk address in the PTW with a null address or not (as the disk error was a per-record error or a device error) before setting ptw.er. When such a page fault is restarted, a successful page fault either pages in zeros or the correct page, respectively.

Write errors determined to be due to an inoperative device cause the posting to cause the modified bit of the page to be turned back on (disk writes only--bulk store cannot be inoperative by this standard), and the core frame to be threaded back in as MRU. This means that the replacement algorithm will reissue the write again when it comes around. If the call side started the write, it calls in again to write it again, as it comes back to see that the page is still modified (or yet modified) when it is notified. Similarly, device-inoperative errors on the write cycle of an RWS cause the PD record not to be freed, but placed back in the PD used list (its modified bit was never turned off), and the free-or-being-freed count (sst.pd_free) decremented. The PD replacement algorithm retries that record at a later time.

The system no longer signals page_fault_error on a read if the cause of the read error is an inoperative device (as opposed to a bad page). This is to avoid signalling errors that might well terminate an absentee process or the initializer in cases where the operator's readying of a disk could allow all software to proceed without error if the supervisor cooperated. Other problematic cases of signalling page_fault_error, such as on a descriptor segment which goes offline during a setfaults operation, are avoided in this way as well.

Instead of signalling page_fault_error, processes that seek to read pages on inoperative devices are made to wait upon a global event, in ring zero, "144163153176"b3, being "dskw" in ASCII, until any disk coming back online notifies this event. The disk DIM performs this notification, and now maintains the bit pvte.device_inoperative, previously used only for drive-test operations, as a copy of its "broken" bit for a given device, notifying this event whenever such a bit is turned off. Any time such a bit is turned on the disk DIM has beeped a "Device requires attention" message to the operator.

The maintenance of pvte.device_inoperative has several implications: when a disk goes off line, the VTOC manager can see that at once, and reject a requested write forthwith, without wasting hot VTOC buffers where not necessary. The create_vtoce primitive can avoid creating segments on inoperative devices. More critically, update_vtoce must be prepared to handle error codes from vtoc_man for inoperative disks, realizing that the vtoce-parts requested were not even put in hot buffers. For this reason, update_vtoce$deact now has an error-code argument.

The implementation of this "disk-offline waiting" feature is facilitated by the fact that all callers of page-reading primitives must obtain the event to be waited on from the primitive in question, because volume-map paging issues preclude any other routine from deducing the wait event. Thus, page reading primitives can now return this global disk offline event, and cause any number of mechanisms to wait and retry on this event. There are exactly three interfaces that call read_page: the page_fault handler, the PL/I-side interface page$pread, and evict_page$abs_wire. These primitives all now check for the presence of ptw.er from a previous read before calling read_page. Thus, if a page read error is posted by the "done" side, an immediate notify causes one of those three interfaces to be reinvoked (via repeated page fault or call-side retry protocol), notice ptw.er, and take special action.

This special action consists of calling page_fault$disk_offlinep to determine if the reason for this error is the disk being offline or some other reason. This is determined by inspecting the PVTE bit set by the disk DIM (there is a window here--it might have been inoperative at one time, but operative now--this is acceptable). If the answer is that the disk is offline, the process page-faulting, call-side (or process-loading-side) reading, or abs-wiring is made to wait on the global disk-offline event. The bit ptw.er is turned OFF at this time, before the process is set waiting, so that when the disk comes back online, a retry of the page fault/reading is made as though no error happened, instead of the detection of the previously set error bit (which this time would be guaranteed to find the disk not inoperative, and thus signal, which is precisely what we are trying to avoid).

If page_fault$disk_offlinep determines that the disk is not offline, an alternate return is made. The page fault handler signals in the way it always used to in this case, and the other two entries just retry desperately and hopelessly as they used to do. (This is the case of a descriptor segment page going bad or similar--an unsolved problem as of this time.).

The call-side wait coordinator, and the notify-requested bit setter in wired_plm (process loading) have been made cognizant about global paging events.

LARGE VOLUME MAP SPACE

In releases 4.0 and 5.0, the single paged unwired segment fsdct held all volume maps. This was an unreasonable space limitation. Volume maps are now in segments fsmap_seg0 to fsmap_seg15, created dynamically by init_pvt at bootload time, as many as are necessary to contain the volume maps for all configured drives. The segment fsdct now contains only what used to be the fsdct header; it is small, unpaged, and wired now.

The code in free_store that returns a PTW pointer and an ASTE pointer to read_page now deduces these quantities from the SDWs of the fsmap_seg, rather than from a fixed pointer in the SST.

Therefore, all references to "FSDCT Paging" in this document should now be read as "Volume map paging".

DAMAGED SEGMENTS

A new VTOC attribute (see Section II, "VTOC Attributes"), thus an ASTE and VTOCE bit, called the "damaged switch", has been introduced (aste.damaged and vtoce.damaged). Although settable and resettable by user-invoked file system calls, the intended function of this bit is to inform the user that page control or the physical volume salvager has either perpetrated or detected damage to this segment. The segment fault handler observes this bit when connecting a process to an ASTE (i.e., constructing an SDW for a segment in a process), and causes "seg_fault_error" with the error code of error_table_$seg_busted to be signalled if it is on. As with other VTOC attributes, the bit is activated and deactivated with the segment., The segment fault handler does not make this check for directories, or in the initializer process (so that the system might always be bootable).

The physical volume salvager and page control construct a standard format binary syserr message (see segdamage_msg.incl.pl1) whenever damage to a segment is created, and log a message with it. This message identifies the segment involved via physical volume ID, LVID, UID, and UID pathname, with other information (e.g., page number) when appropriate.

The physical volume salvager constructs this information from a VTOCE being processed, the UID pathname being copied from the third vtoce-part. Page control deduces it from AST entries, chasing up the AST parent pointers to develop the UID path (this logic is in the module page_error). The physical volume salvager "damages" segments whenever any VTOCE inconsistency is discovered: the case where segment control deliberately introduces an inconsistency during VTOCE update before a fatal crash is particularly important here. Page control damages a segment when a disk error on reading or writing occurs that is due to a bad record as opposed to a bad device.

The counter sst$damaged_ct is incremented whenever such a binary message is logged. The answering service's accounting-update metering program (as_meter_) inspects this variable at each accounting update. If it has increased (since the last update, or bootload time, initially), the syserr log is scanned for such messages. They are read out, the UID pathnames in them converted to ASCII pathnames, and the interpreted messages logged in the answering service log.

QUOTA VALIDATOR

Reimplementation of what had been the salvager in release 5.0 and earlier, for this release removed the function of computing quota-used from it. Quota-used computation was the only part of the salvager that could not be done by a top-down hierarchy scan; one cannot compute correct quota-used for a directory until correct quota-used totals have been computed for inferior directories; this severely limits the implementation flexibility of salvaging functions. What is more, the algorithms up to now for correcting quota-used required the entire hierarchy to be quiescent: thus crashes for which ESD has failed (almost guaranteed to create quota-used inconsistencies see below), required a "long salvage" while no one was logged in (the only way the salvager could be run).

The discovery of an algorithm to compute correct quota-used totals in a nonquiescent hierarchy has obsoleted all of this, and is now the only way that quota-used is corrected. The hierarchy salvager is now nothing more than a program that reformats a single directory, optionally cross-checking VTOCEs. Conventional ring-4 programs are used to map the salvager over subhierarchies. Quota and quota-used are now out of its domain.

In order to understand the on-line correction algorithm, it is necessary to understand how quota-used inconsistency arises. A subhierarchy is said to have inconsistent quota-used if any directory in it has inconsistent quota-used. A directory is said to have inconsistent quota-used if its quota-used figures (for segments or directories) are anything but what they should be. The (directory or segment) quota-used figure of a directory should be the sum of the (directory or segment) quota-used figures of all immediately inferior directories that do not have terminal (directory or segment) quota accounts, plus the sums of the records-used of all immediately inferior directories or segments. This is dependent upon all subhierarchies being quota-used consistent.

A directory becomes quota-used inconsistent in the following way: a segment is deleted or some pages are created. Several directories have their quota-used figures adjusted by page control (in the ASTE) at the time this happens. At some later time, the VTOCE for one of the directories is updated; perhaps the lower one is deactivated, or the AST trickle updates one of them. The VTOCEs now reflect an inconsistent quota-used situation, for the VTOCE of one directory claims records charged to it, but the other does not. If the system shuts down successfully there is not problem, as all VTOCEs are updated. Before the system shuts down, anyone who wants to know the quota-used figures goes to VTOCE or ASTE as appropriate, and the inconsistency of the VTOCEs is not a problem. However should the system crash and not shut down, the next bootload relies solely on VTOCE information, and a quota-used inconsistency results.

It may be seen that quota-used inconsistency is not the result of a supervisor malfunction, but rather a misfeature of fatal (no ESD) crashes. They are a consequence of not stopping the entire system to update disk-resident data every time a page is created or destroyed. Quota-used inconsistencies do not develop while the system is running.

The online correction algorithm is based upon the fact that quota used for a given directory is either right or wrong at any time. If it is right to start with, it cannot go wrong while the system runs. If it was wrong to start with, the amount by which it is wrong is a constant from the time the system was bootloaded to the time it is fixed. It cannot get more or less wrong by its own volition.

The task of the quota validator is thus to determine exactly how much (if at all) a given quota-used figure is wrong and fix it. It can fix it at any time after it determines by how much it is wrong--a certain number is to be added or subtracted. The quota-used figure is not just replaced.

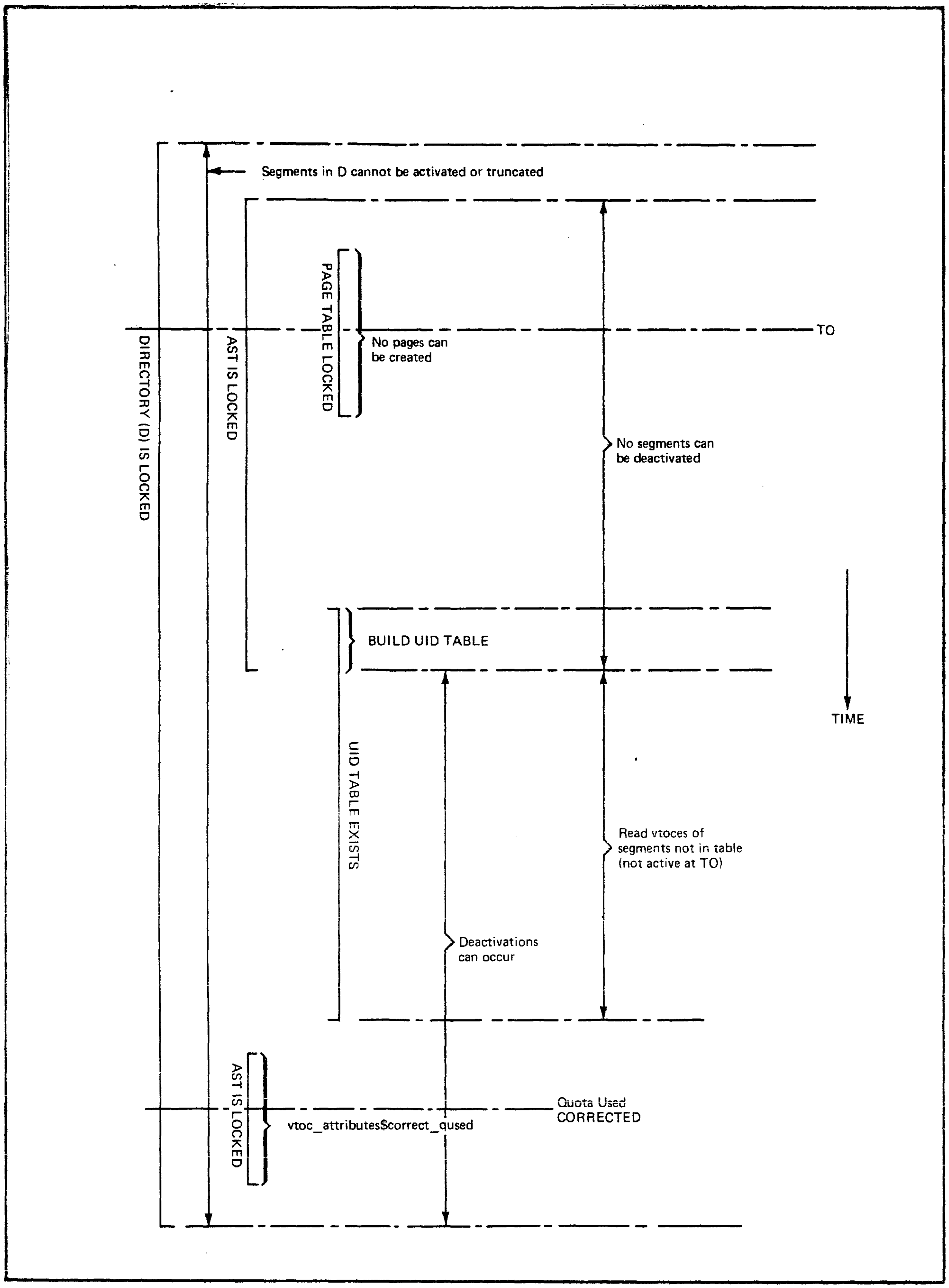


Figure A-2. Quota Validator

To understand this more fully, hypothesize that there were a tool available that corrected quota used, say set_quota_used <dirname>. A system administrator might want to figure out the correct number, and set it. However, this would be inordinately difficult to use, because even the wrong number is constantly changing. Thus, the only kind of tool that would be of value is one that added or subtracted its argument from the quota-used figure, regardless of what it was--a tool that added or deleted phantom segments.

The quota validator operates precisely in this way. The entry vtoc_attributes$correct_qused performs precisely the function of adding a signed difference to a quota-used total for a directory, either in a VTOCE or in an ASTE, once the correct difference has been determined. The determination of the value of this difference is a very intricate operation, involving several locking games. We can approach this algorithm by successive refinement.

Given our choice, we would quiesce the entire subhierarchy of the directory (which we will call D) whose quota-used is being computed. We would lock the page-table lock and the AST lock, read all the VTOCEs and AST entries for immediately inferior segments and directories, adding their page totals and quota-used figures (for directories), from the AST for active segments and from the VTOCEs for nonactive segments. Comparing that total to the current quota-used gives us the difference we seek. However, we cannot randomly go locking locks like that, or quiesce the subhierarchy in this way. We therefore choose one moment in time for which we will strive to compute D's correct quota-used total. For any given instant, we can quiesce all of page control activity (creating and deleting pages of active segments in particular) by locking the page table lock. Call that instant T0. We choose such an instant, and lock the page table lock before it. At that instant, with the page table lock locked, we compute the sum of the records-used totals of segments immediately inferior to D, that subset of them that is active at T0, plus the sum of the quota-used figures of immediately inferior (nonterminal) directories, that subset of them that was active at T0. This figure is an approximation to the correct sum of records-used plus inferior quota-used for this directory at T0. It is inaccurate by precisely the sum of the records-used plus nonterminal quota used of exactly that set of immediately inferior segments and directories that were not active at T0. Thus, once the page table lock is unlocked, we need only add up the figures for these segments. However, we do not wish to read all the VTOCEs, or scan D with the page table locked. If we unlock the page table lock, other segments may be activated or deactivated, and we would have no way or determining which segments were active at T0 and which were not.

Pages are created only by touching them, and since only pages of active segments can be touched, no pages can be created for inactive segments if we prevent them from being activated. Similarly, pages can be destroyed by two means: manipulations on active segments (truncations, page zeroings), and file-system calls (truncate, delete) on inactive segments. Thus, if we prevent new segments in D from being activated between T0 and the time quota-used of D is corrected, and prevent file system operations on segments in D in this interval at well, we can be sure that the quota-used subtotal for segments inactive at T0 will not change between T0 and the time quota-used of D is corrected. It turns out that locking D prior to the start of this whole operation accomplishes precisely this.

With this in mind, we know that no segments that were not active at T0 can be activated after the page table lock is unlocked. What is more, they cannot be otherwise affected (e.g., truncated). So at this stage of development, our algorithm is to unlock the page table, scan D, and check each segment in it for activity at time T0 (it couldn't be active now if it wasn't active then) and add its quota-used or records-used to the total from time T0. This does not work because segments can get deactivated between the time the page table lock was unlocked and the time we check the AST to see if it was counted in the total at time T0. Segments can be prevented from being deactivated by having locked the AST after first locking D, but before locking the page table lock. Thus, when we scan D, the AST will still be locked, and the set of active inferiors of this directory will not have changed since time T0.

The deficiency here is that one may not touch a directory with the AST locked (see the general considerations of the locking algorithm in "Locking Conventions", Section II). To determine which segments were active at time T0 we lock the AST lock before locking the page table lock, and unlock the AST lock after unlocking the page table lock. But before unlocking the AST lock (at a time when the set of active segments cannot have changed since T0), we build a little table of the UIDs of all active inferiors of this directory in automatic storage. It is with this table that we check while scanning the directory adding up quota and records figures from VTOCEs.

Having added the active and inactive figures, they are compared with the value of the quota-used figure of this directory read at time T0 to determine the finite and invariant error. It is this error that is deducted from the quota-usecd figure of D.

This algorithm is implemented in the program correct_qused. The entry quotaw$rvq performs the manipulations and quota cell readings under the page table lock.

The bottom-up walking features of do_subtree (or walk_subtree) are used to drive the tool fix_quota_used (the ring 4 interface to the quota validator) bottom-up.

SUPPORT OF HIERARCHY SALVAGER

The mechanism used by the hierarchy salvager to activate, deactivate, and access segments, dating from the time that the salvager had its own tape, is entirely gone in release 6.0. The entire activation/file map mechanism described in "Services on Behalf on the Hierarchy Salvager" in Section IV has been removed. The hierarchy salvager is now a directory-control program that operates on one directory at a time, given its pathname. It initiates directories and takes segment faults upon them, as any other directory control program in Multics. It has no more involvement with segment control. The removed interaction with segment control had been a major source of bugs.

The central control program of the hierarchy salvager, salv_directory, is usually driven by ring-4 subtree walk. It does not recurse.

The hierarchy salvager no longer uses abs-segs or any abs-seg mechanism; it no longer checks, validates, or corrects quota or quota-used.

The hierarchy salvager retains a "VTOCE-checking" feature, used to check for (forward) connection failure, optionally delete branches suffering this, and correct part III (permanent attributes) information. These functions are provided in the program salv_check_vtoce, which is not even called if VTOCE checking was not specified to the hierarchy salvager. The program salv_check_vtoce calls vtoc_man$get_vtoce to obtain a VTOCE image, to check UID match and part III information. If information need be corrected in the VTOCE, the entry "salv_update" in vtoc_attributes is called to correct and write back information to be updated. As usual, vtoc_attributes is cognizant of all rules regarding directory and AST locks for such operations (see Section II). Thus, the hierarchy salvager no longer directly writes VTOCEs in any case.

To delete branches suffering forward connection failures, salv_check_vtoce calls a special entry in directory control's "delentry" primitive, that which deletes branches.

The hierarchy salvager makes use of the grab_aste/prewithdraw mechanism described above and in Section IV to cause semi-permanent activation of its scratch and directory-copy segments.

LIMITED UPDATE BACKLOG

The 6.0 storage system tries to enforce an upper bound on the time the AST trickle takes to circumnavigate each AST used list (see Section II). By placing an upper bound on this time, file map changes cannot stay in the AST (not be reported to the VTOCE) for longer than this maximum time. This is done solely as a hedge against fatal crashes under light load. In these cases, it has often been reported that a segment modified hours before the crash appears empty (all zeros) at the next bootload. This was because of failure to update its VTOCE within a reasonable period of time. in release 6.0, the initializer calls into get_aste$flush_ast_pool with a pool index every accounting update if it has been determined that fewer steps in that pool than the number of entries in it were taken since the last such update (the accounting update routinely inspects meters in the SST). The entry get_aste$flush_ast_pool circumnavigates the specified AST list one entire time, calling update_vtoce on each ASTE whose file map has changed (aste.fmchanged). This fairly expensive action is invoked if and only if load is so light that there was not a reasonable number of AST steps in the last accounting interval.

A similar attempt is made to set an upper bound on the amount of time a page may stay in main memory and not be written out. This, again a hedge against fatal crashes, is to guard against the phenomenon where a heavily-modified page remains in memory under light load, and does not get written out, and appears zero or nonexistent at the next bootload. A page is written out if load is light, i.e., the circulation speed of sst.usedp is slow, and continual use and modification biases the replacement algorithm against writing this page out.

The new entry pc$flush_core, and the new CME bit cme.phm_hedge implement this facility. The entry pc$flush_core is called five minutes before every accounting update (by the initializer) to call page$pwrite on all pages not written out since the last such call. The five-minute interval is to make sure that the accounting update that follows, calls get_aste$flush_ast_pool, is able to report new page creations to VTOCES, i.e., to ensure that writes started complete successfully before VTOCE updating is attempted (see "Address Management Policy" in Section VII for why the VTOCE can't be updated until successful completion of writes is acknowledged). The entry pc$flush_core scans the core map for all in-core pages that need to be written out, and calls page$pwrite, multiplexing activity in the normal page control manner (see Sections VIII and IX). These pages are identified by the presence of the flag cme.phm_hedge. This bit is turned on by pc$flush_core for every in-core page having ptw.phm on, that it is not calling page$pwrite to write out. Page control (page$pread and the "write" side of the interrupt side, page$done_) turn this bit off any time a page is read into this frame, or a successful write is completed from it. Thus, if pc$flush_core finds (the next time it is called) that this bit is still on, it can deduce that this frame had a modified page in it one accounting interval ago, and has not been evicted or written out since. This is precisely the condition for issuing a write for the page in that frame.

PARTIAL SHUTDOWN

Page and segment control primitives called at shutdown time (Emergency or Regular) have been changed to check the PVTE bit pvte.device_inoperative before attempting to update a VTOCE (including calling pc$cleanup), flush a main memory page or initiate an RWS. All drives are tested at the time shutdown is started (earlier still in ESD), in the procedure disk_emergency (in bound_disk_util_wired). By calling the standard drive-testing primitive (read_disk$test_drive, see "Explicit Disk Reading, Writing, and Testing" in Section XIV) the operative/inoperative status of all drives is determined. What is more, the interrupt sides of page control and of the VTOC manager call an entry in disk_emergency which evaluates whether or not to set pvte.device_inoperative whenever they receive a "device-inoperative"-type error from the disk DIM. The program disk_emergency sets the bit only during shutdown; otherwise, the disk DIM maintains it. At shutdown time, disk_emergency also notifies the Operator about disks which went offline during (or before the start of) shutdown.

Thus, during shutdown, all drives not inoperative are completely shut down. The complete shutdown of the RPV is not indicated unless all other drives were shut down; this is to force a hardcore directory salvage and paging device flush on the next bootload. All packs not shut down will be salvaged the next time they are accepted, as is usual.

The code and variables of Emergency Shutdown have been so reorganized that ESD may be attempted any number of times after a partial shutdown, if drives can be brought back up. If the drives have indeed become operative (all drives are tested afresh each time), a completely successful ESD will be attained. Unflushable contents of the paging device and main memory will be kept around until this is the case.

The avoidance of complete shutdown of the RPV causes the next bootload to take cognizance of the unflushed paging device, which is necessary.

OTHER CONSIDERATIONS

In Section VI, cme.devadd now points to the PDME during the entire RWS.

The variable "did" in pxss_page_stack (the ALM page control environment stack frame) has justly and finally been renamed "pvtx", which is what it had meant since release 4.0.

A fairly baroque error-message generating facility has been built into page_error.alm, taking advantage of the new macro processor in the ALM assembler. Incorporated in this facility is the logging of binary syserr messages indicating segment damage. The page_error program includes a system of macros for declaring variables and generating PL/I-like calls automatically, and is worth investigation by those interested in ALM or assembler technology.

In Section VIII, the "second trace facility", or "disk_meters" has been totally removed.

The subroutine cleanup_page is now the only agency in the system (outside of pc_recover_sst, that is a highly special case) that evicts pages. The routines in pc.pl1 have been changed to call it, as page$pcleanup. Consistency required by pc_recover_sst motivated this change.

In Section X, some reorganization of utility subroutines, particularly in pd_util, was performed.

A (privileged) user-callable facility to entry-hold a segment and wire its pages via calls to pc_wired has been provided.

The updating of time-page product to a directory's parent at the time of its deletion was found to be lacking in Releases 4.0 and 5.0. This function was added in delete_vtoce, which, in the case of a directory with terminal quota being deleted, performs several VTOCE manipulations under the AST lock to update this VTOCE-resident quantity from the directory being deleted up.

Reused and unprotected disk addresses, as well as bad VTOC threads, no longer cause the system to crash. Volumes suffering these symptoms are placed in a state (pvte.nleft = 0) where no new allocations can take place on these volumes, and scheduled for salvage (pvte.vol_trouble = "1"b). These new policies are due to a belief in the current stability of the supervisor: that such symptoms can not occur as a result of a software malfunction in the current bootload, but are more likely symptoms of disk malfunction or bad data from a previous bootload.

The "PD Writeahead" experimental feature has been removed.

The disk record allocator has been recoded to be more straightforward: the remnants of older schemes have been replaced by code which has the same effect, but by explicit design.

The disk-reading primitive (read_disk, Section XIV) is now used by volume backup in many processes, and thus can no longer use the unshareable supervisor ASTE (PTW-level abs-seg) read_disk_seg in all processes. It continues to use this ASTE if running in the initializer process, initialization, or shutdown. in any other process, an ASTE is gotten via normal means (get_aste) to use a an abs_seg.

The VTOC attribute array for record quota (aste.quota, vtoce.quota) is redefined as seg_vtoce.usage_count and seg_aste.usage_count, a count of page faults on a segment maintained by page control, for nondirectory segments. A file system call through mhcs_ is available to obtain this VTOC attribute. It is not in hcs_ because the observing of this datum constitutes an AIM write-down path, and discretionary access control to this meter may be desired at some AIM sites.

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS STORAGE SYSTEM
PROGRAM LOGIC MANUAL
ADDENDUM A

ORDER NO. AN61A, REV. 0

DATED SEPTEMBER 1978

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. □

FROM: NAME ______________________________ DATE ____________

TITLE ______________________________

COMPANY ______________________________

ADDRESS ______________________________

____

CUT ALONG

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

18684, 7.5C877, Printed in U.S.A.