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

ILLUSTRATIONS

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 <u>this</u> ASTE is the parent. Like all ASTE lists and pointers, it is <u>zero</u> 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 commence 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.

TRAILERS AND SETFAULTS

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 trailer mechanism, and the operations known as setfaults, 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 off, 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 segment fault. 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 all SDWs pointing to a given segment must be revoked (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 trailer list 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 not, 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 not 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.

VTOC BUFFER SEGMENT

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 n 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 progress, 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 <u>not</u> 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 all 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 not 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 percentage 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 deposited (placed in the free storage pool for that pack, by calling pc$deposit_list) until it is known for a fact that the VTOCE from which they were removed has been successfully written out to disk. Were this not so, it would be possible that some addresses might be deposited, picked up by a new segment, and written out to that VTOCE. Then, if the VTOCE which had the addresses originally was not yet successfully written out, or badly written out, and the system crashed at that point, two VTOCEs would both contain the same record address, a situation known as a "reused address" which is a bad security violation. Thus, the primitive in the VTOC manager, vtoc_man$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 <u>has</u> 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).

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, '00PV'") 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.