SECTION V

PAGE CONTROL OVERVIEW AND CONCEPTS

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

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

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

Section V. Overview and Concepts, the current section, explaining basic concepts and goals of page control.

Section VI. Data bases, breaking down the fundamental data objects of page control, the PTW, the CME, the PDME, the PDMAP header, and the free store maps in the PVTE/FSDCT.

Section VII. The address management policy used by Multics to avoid accidental disclosure of data by virtue of inconsistencies and crashes.

Section VIII. The fundamental mechanisms and protocols used within page control to support the services provided.

Section IX. The services provided by page control to Multics, explained in terms of the mechanisms and data bases described in Sections VI, VII, and VIII.

Section X. Peripheral services of page control.

Section XI. Quota management.

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

BASIC GOALS AND SERVICES OF PAGE CONTROL

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

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

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

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

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

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

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

constants require only a subset of all processes eligible to run at any time. The traffic controller gives processes eligibility and takes it away depending on scheduling decisions; a process that is eligible cannot run until it is loaded. This loading consists of wiring its two critical pages. Similarly, when eligibility is taken away, a process is unloaded. The loading of processes is initiated immediately at the time the traffic controller makes them eligible. The service of loading and unloading processes for the traffic controller is an important auxiliary service of page control.

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

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

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

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

BASIC ORGANIZATION OF PAGE CONTROL

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

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

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

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

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

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

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

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

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

PAGE TABLE LOCK

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

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

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

OUTLINE OF THE DATA BASES OF PAGE CONTROL

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

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

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

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

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

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

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

The first few records of the bulk store are not used as part of the paging device; rather, the paging device map is written out from main memory to as many of these first few records as need be to contain it, every second. This is done as a hedge against fatal (no ESD) crashing. Should the system crash unrecoverably, the next bootload can read the contents of the first few records of the bulk store, and obtain the old paging device map, accurate to within a second. As physical volumes are accepted (see Section XII) by that next bootload, pages of segments on that volume are repatriated from the old paging device contents as their VTOCEs are processed by the physical volume salvager. A Unique ID and page number are put in each paging device map entry to facilitate repatriation; because of these two quantities, the second inaccuracy of the paging device map need not be a cause for concern. Thus, the paging device map has potentially a cross-bootload longevity. To facilitate

interpretation of its contents, the PDMAP (as the paging device map is sometimes called, not to be confused with sst.pdmap, which stands for paging device map array pointer) has a four-word header, the pdmap header, describing the extents and time of initialization (called the PDMAP time) of the paging device map. This PDMAP time is marked in the volume labels of all physical volumes which were part of the configuration during which that PDMAP was used; this is the key to the mechanism (explained fully in Section VIII, under "Post-Crash PD Flush") by which pages are repatriated as volumes are accepted. Because the first record of the bulk store contains the first page of the PDMAP, the first PDME of a PDMAP is not used, but contains the PDMAP header. All PDMEs that describe records similarly used by the PDMAP image other than the first are not used at all, and contain all zeros.

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

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

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

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

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

ZERO PAGES

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

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

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

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

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

MAIN MEMORY REPLACEMENT ALGORITHM

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

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

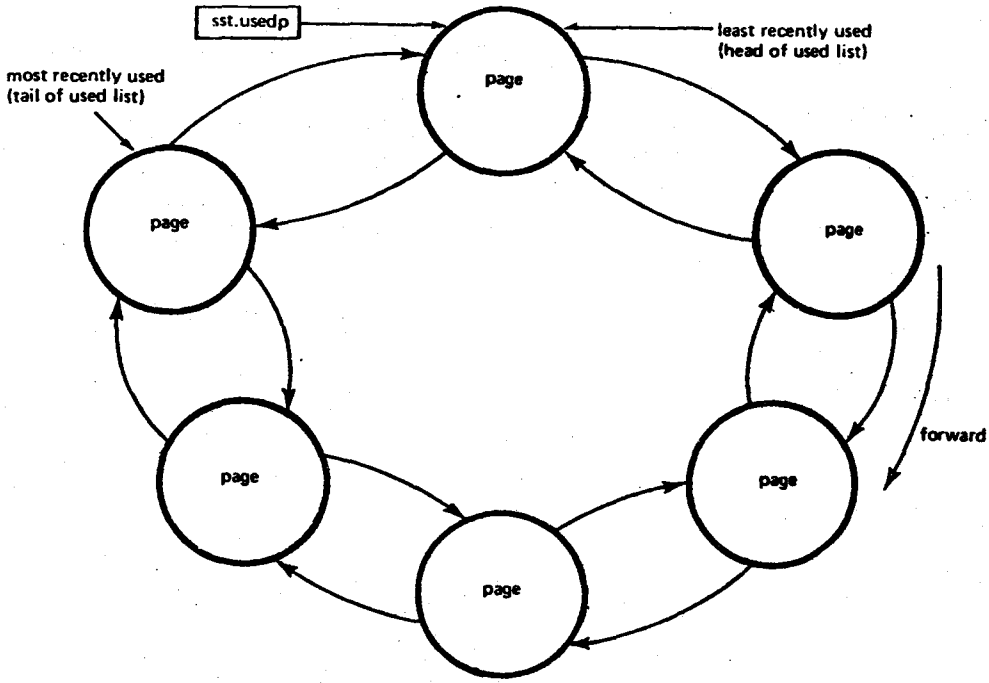


Figure 5-1. The Clock Algorithm

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

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

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

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

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

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

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

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

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

PAGING DEVICE MANAGEMENT ALGORITHM (PAGE MULTILEVEL)

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

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

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

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

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

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

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

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

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

The paging device replacement algorithm is invoked at the beginning of page fault processing, every page fault. It tries to ensure that a small, fixed number (10) of paging device records are always free or in the process of being freed (RWS in progress). Since it does this at the beginning of a page fault, when it is finished, probably some paging device records will have been freed, some already free, some started RWSs, and some finished RWSs from some previous time (made free by the interrupt side). Thus, it is probabilistically very likely that some records will be free during the processing of that page fault (during which claim_mod_core may attempt to migrate pages to the paging device). The replacement algorithm moves down the PD used list, evicting all pages not requiring RWS, and starting RWSs for all pages modified with respect to

secondary storage. PD records found to contain pages that are also in main memory are rethreaded in the list so that they acquire the favored "recently seen to be used" status. This action continues until ten records are free or in RWS. There is no problem of obtaining "RWS buffer" pages here, a call being made to find core as each such buffer is needed. Note that find_core will not cause PD records to become allocated in so doing; find_core does not initiate writes. Only claim_mod_core does that.

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

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

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

Papers about the Multics Page Replacement Algorithm:

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

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

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

SECTION VI

PAGE CONTROL DATA BASES

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

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

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

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

PAGE CONTROL DEVICE ADDRESS (devadd)

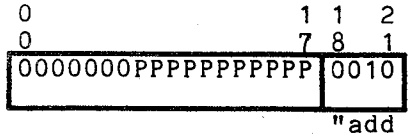
One quantity that crops up in PTWs, CMEs, and PDMEs is the general device address. A device address designates a frame of main memory, a record of paging device, or a record of disk. A device address, or devadd, has two subfields, the address, or record address, as befits which of the above cases is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

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

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

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

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

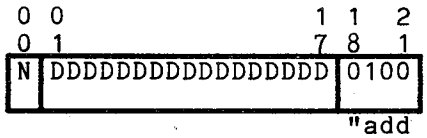


"add_type," here add_type.pd

PPP = paging device record number.

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

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



"add_type," here add_type.disk

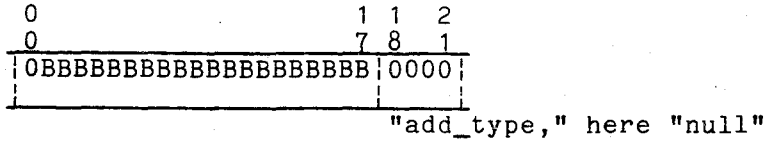
DDD = Disk record number.

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

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

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

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

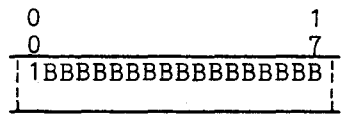


BBB = debugging code.

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

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

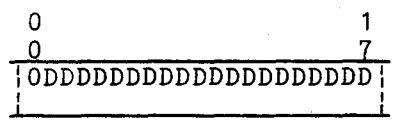
Format of a segment control, or file map null address, never valid in page control, only valid in file maps in VTOCEs:



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

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

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



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

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

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

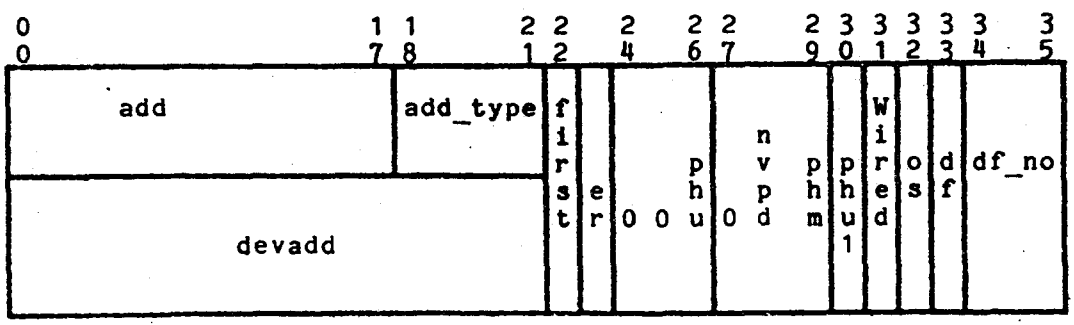
2. The core map entry (CME), representing a page-frame of main memory and describing its association, if any, with any page of any segment.

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

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

PTW, OR PAGE TABLE WORD

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

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

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



ptw.add
(777777,du)

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

ptw.add_type
(740000,dl)

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

mptw.devadd
(777777740000)

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

ptw.first
(200000,dl)

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

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

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

ptw.phu
(001000,dl)

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

ptw.nypd
(000200,dl)

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

ptw.phm
(000100,dl)

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

ptw.phu1
(000040,dl)

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

ptw.wired
(000020,dl)

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

ptw.os
(000010,dl)

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

ptw.df
(000004,d1)

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

ptw.df_no
(000003,d1)

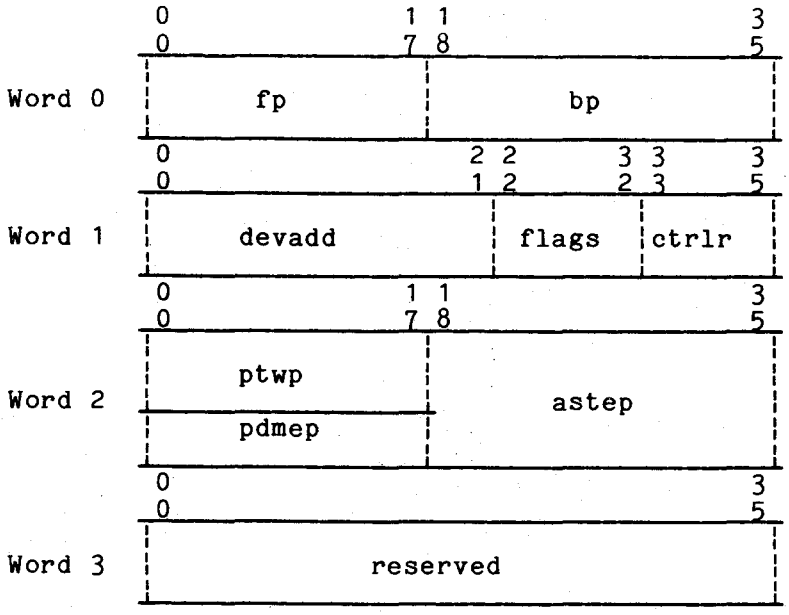
The contents of this field tell the hardware what type of directed fault to take when ptw.df indicates that it should take a fault. In Multics, this field is always set to "01"b, and thus, a directed fault 1 is interpreted as a Multics page fault. Note that zeros in a PTW, or an attempt to use zeros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

CORE MAP

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

CORE MAP ENTRY (CME)

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

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

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

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



cme.fp
(777777000000,word 0)

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

cme.bp
(000000777777,word 0)

Back pointer. See cme.fp above.

cme.devadd
(777777740000,word 1)

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

cme.flags
(000000037770)

Various state flags, detailed below.

cme.io
(004000,d1)

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

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

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

cme.rws
(002000,d1)

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

cme.er
(001000,d1)

is NOT USED.

cme.removing
(000400,d1)

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

cme.abs_w
(000200,d1)

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

cme.abs_usable
(000100,d1)

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

cme.notify_requested
(000040,d1)

Valid only if cme.rws is off, and cme.ptwp describes a CME with ptw.os on (in which case this CME is threaded out of the used list, as a page I/O is in progress). Tells the interrupt side that some process is waiting, via the traffic controller wait/notify mechanism for I/O completion on this page. This bit is turned on when any

process goes to wait for paging I/O, either on the fault side (see "Page Fault Handling" in "Services,") the call side, via the call-side wait coordinator, device_control$pwait (see "Wait Protocols" in "Mechanisms"), or the special wait mechanism of the process-loading mechanism (see "Process Loading" in "Services"). It tells the interrupt side to invoke the traffic controller to perform a "notify" on the event associated with this page (see "Wait Protocols" in Section VIII) when the I/O on this page is complete. If not on, no traffic control notify is performed when this I/O completes.

cme.pd_upflag
(000020,dl)
Causes the interrupt side to rethread this CME to most recently used position on the completion of a page write from this frame, as opposed to the least recently used position as it normally does.

cme.contr

(000007,dl)
Not currently used. (Controller) is the port tag of the system controller that controls the main memory described by this CME. (See the Multics Reconfiguration PLM, Order No. AN71.)

cme.ptwp
(777777000000,word 2)
PTW pointer. Only valid when cme.rws is off. When nonzero, states that some page of some segment is associated with this page frame. The field cme.ptwp is the relative offset into the SST of the PTW for that page. The page may or may not be undergoing I/O as ptw.os of that PTW is on or off. The page is not, however, undergoing RWS. It is guaranteed that the "devadd" file of the PTW has a main-memory type devadd describing the main memory page frame of this CME.

mcme.pdmep
(777777000000,word 2)
(Alternate for cme.ptwp). Only valid when cme.rws is on, which is when there is an RWS going on in this main memory frame. In this case, mcme.pdmep is the relative offset into the SST of the PDMAP entry of the PD record undergoing this RWS. In this case, the field mpdme.cmep of that PDME would be the relative offset into the SST of this CME.

cme.astep
(000000777777,word 2)
Only valid under the conditions under which cme.ptwp is valid and nonzero. The field cme.astep will then contain the relative address into the SST of the AST entry for the segment to which the page in this main memory frame belongs.

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

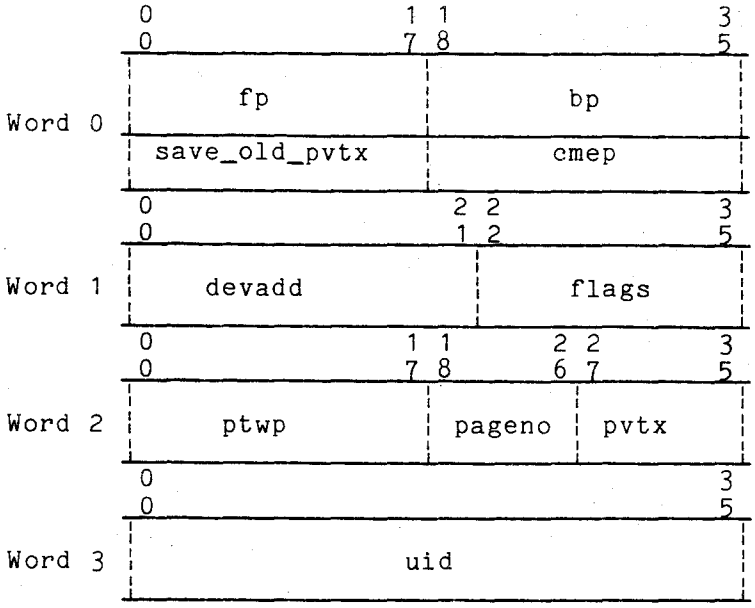
PAGING DEVICE MAP

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

PAGING DEVICE MAP ENTRY (PDME)

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

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

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

    2 uid bit (36) aligned;

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

This page intentionally left blank.



pdme.fp
(777777000000,word 0)

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

mpdme.save_old_pvtx
(377777,du,word 0)

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

pdme.bp
(000000777777,word 0)

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

mpdme.cmep
(000000777777,word 0)

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

pdme.devadd
(777777740000,word 1)

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

pdme.flags
(037777,d1,word 1)

Are the pdme control flags, detailed below.

pdme.mod
(004000,d1)

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

pdme.incore
(002000,d1)

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

pdme.rws
(001000,d1)

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

pdme.used
(000400,d1)

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

pdme.abort
(000200,d1)

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

pdme.flushing
(000040,d1)

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

pdme.notify_requested
(000020,d1)

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

pdme.update_only
(000010,d1)

Is OBSOLETE.

pdme.removing
(000004,d1)

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

pdme.double_writing
(000002,d1)

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

pdme.ptwp
(777777000000,word 2)

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

pdme.pageno
(377000,d1)

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

pdme.pvtx
(000377,dl)

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

pdme.uid
(whole word 3)

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

PDMAP HEADER

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

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

pdmap_header.pd_first

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

pdmap_header.pd_using

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

pdmap_header.nrecs_pdmap

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

pdmap_header.pdme_no

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

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

PVTE VARIABLES FOR PAGE CONTROL

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

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

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

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

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

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

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

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

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

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

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

SYNOPSIS OF RELEVANT SST VARIABLES

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

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

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

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

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

sst.npfs_misses
is OBSOLETE.

sst.salv
is OBSOLETE.

sst.ptl
is the actual global page table lock.

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

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

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

sst.astsize
is 12 decimal, the size of an AST entry.

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

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

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

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

sst.startp
is OBSOLETE.

sst.removep
is OBSOLETE.

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

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

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

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

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

sst.nolock
is OBSOLETE.

sst.x_fsdctp
is OBSOLETE.

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

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

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

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

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

sst.pc_io_waits
is OBSOLETE.

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

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

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

sst.ctwait
is OBSOLETE.

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

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

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

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

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

sst.skipos
is OBSOLETE.

sst.skipspd
is OBSOLETE.

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

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

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

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

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

sst.pre_page_size
is OBSOLETE.

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

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

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

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

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

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

sst.wired_copies
is OBSOLETE.

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

sst.first_core_block
is zero.

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

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

sst.pp_meters
is OBSOLETE.

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

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

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

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

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

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

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

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

sst.claim_steps
is a count of core map entries processed by claim_mod_core.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

sst.pdhtp
is OBSOLETE.

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

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

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

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

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

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

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

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

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

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

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

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

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

sst.pd_skips_rws
is OBSOLETE.

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

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

sst.pd_rws_active
is OBSOLETE.

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

sst.pd_read_truncates
is OBSOLETE.

sst.pd_write_truncates
is OBSOLETE.

sst.pd_htsize
is OBSOLETE.

sst.pd_hash_mask
is OBSOLETE.

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

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

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

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

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

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

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

sst.pd_time_counts
is OBSOLETE.

sst.pd_time_values
is OBSOLETE.

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

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

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

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

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

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

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

sst.buckets
: (See sst.count_pdmes.)

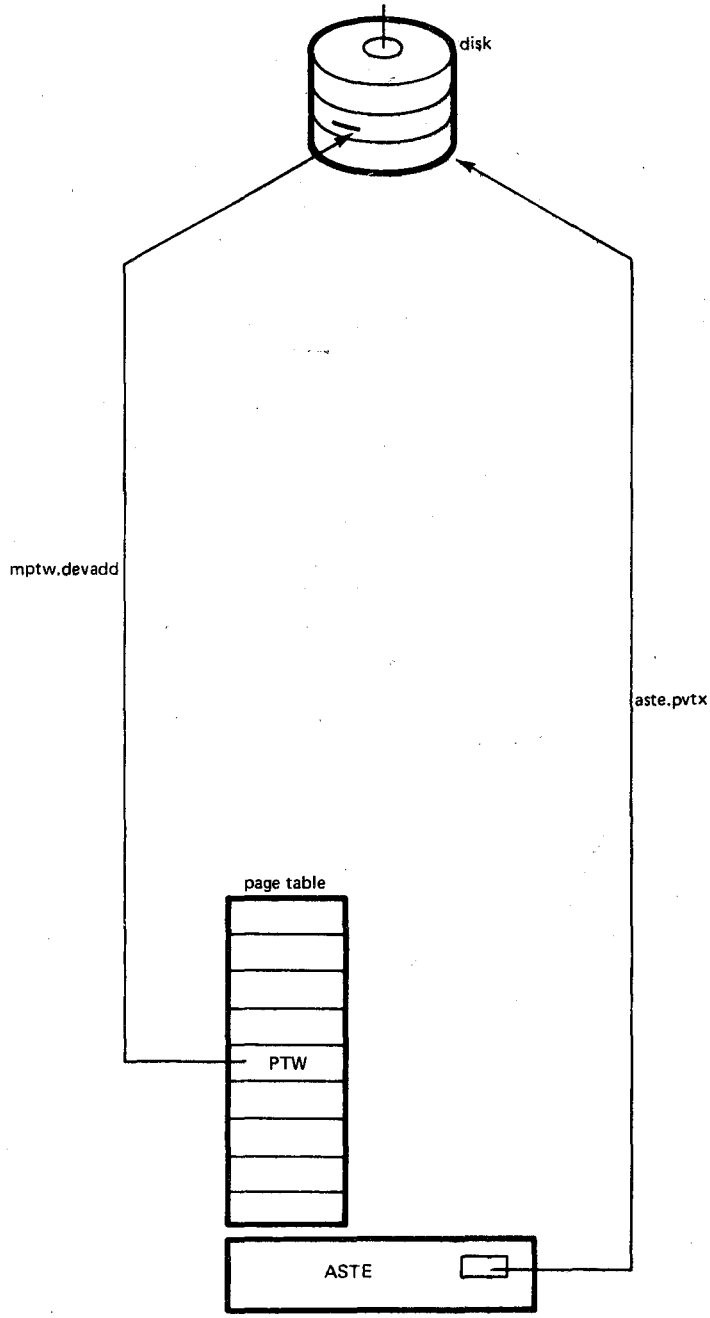


Figure 6-1. Page Control Data Bases
Page not in main memory or on paging device

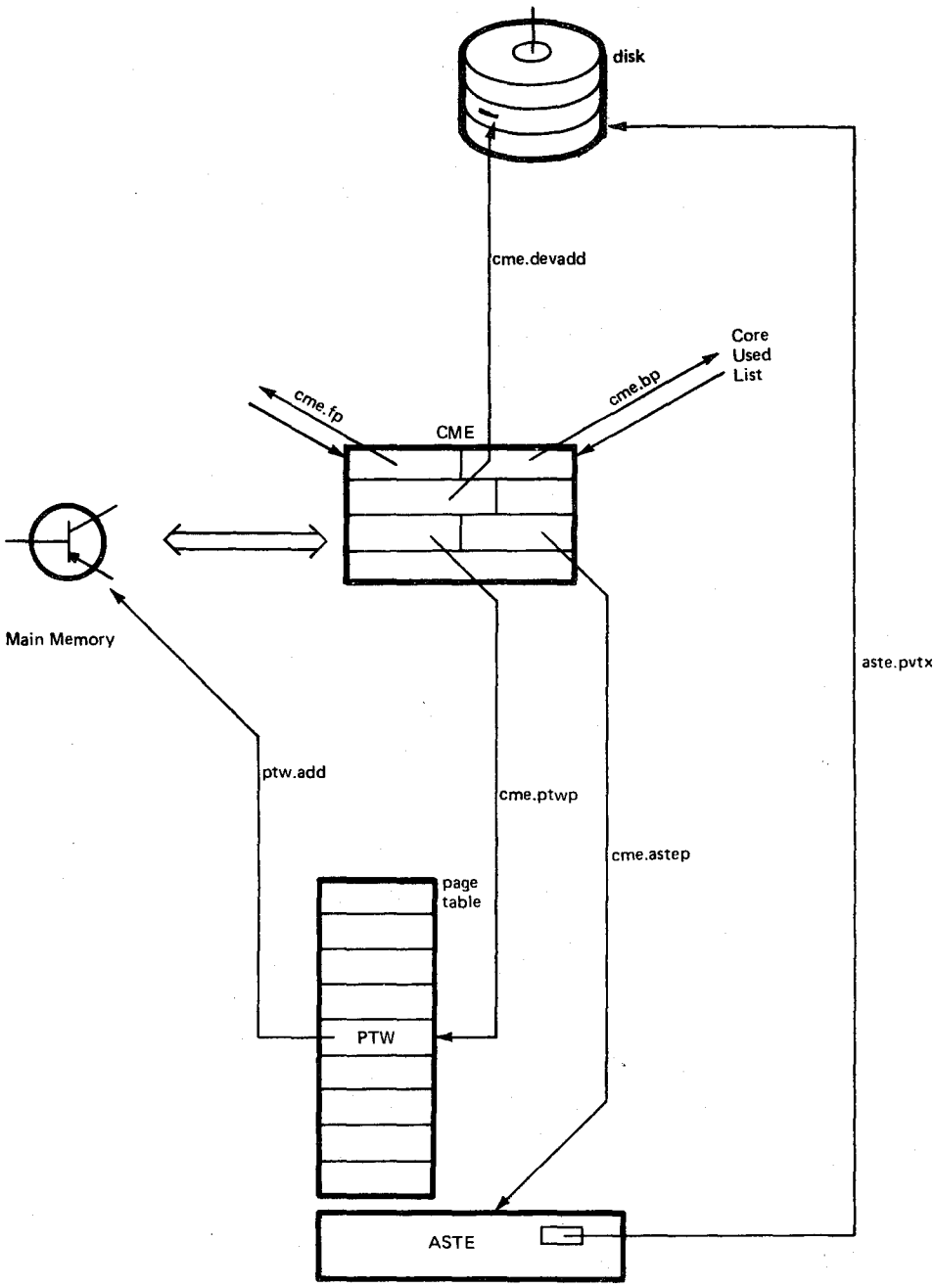


Figure 6-2. Page Control Data Bases
Page in main memory, not on paging device

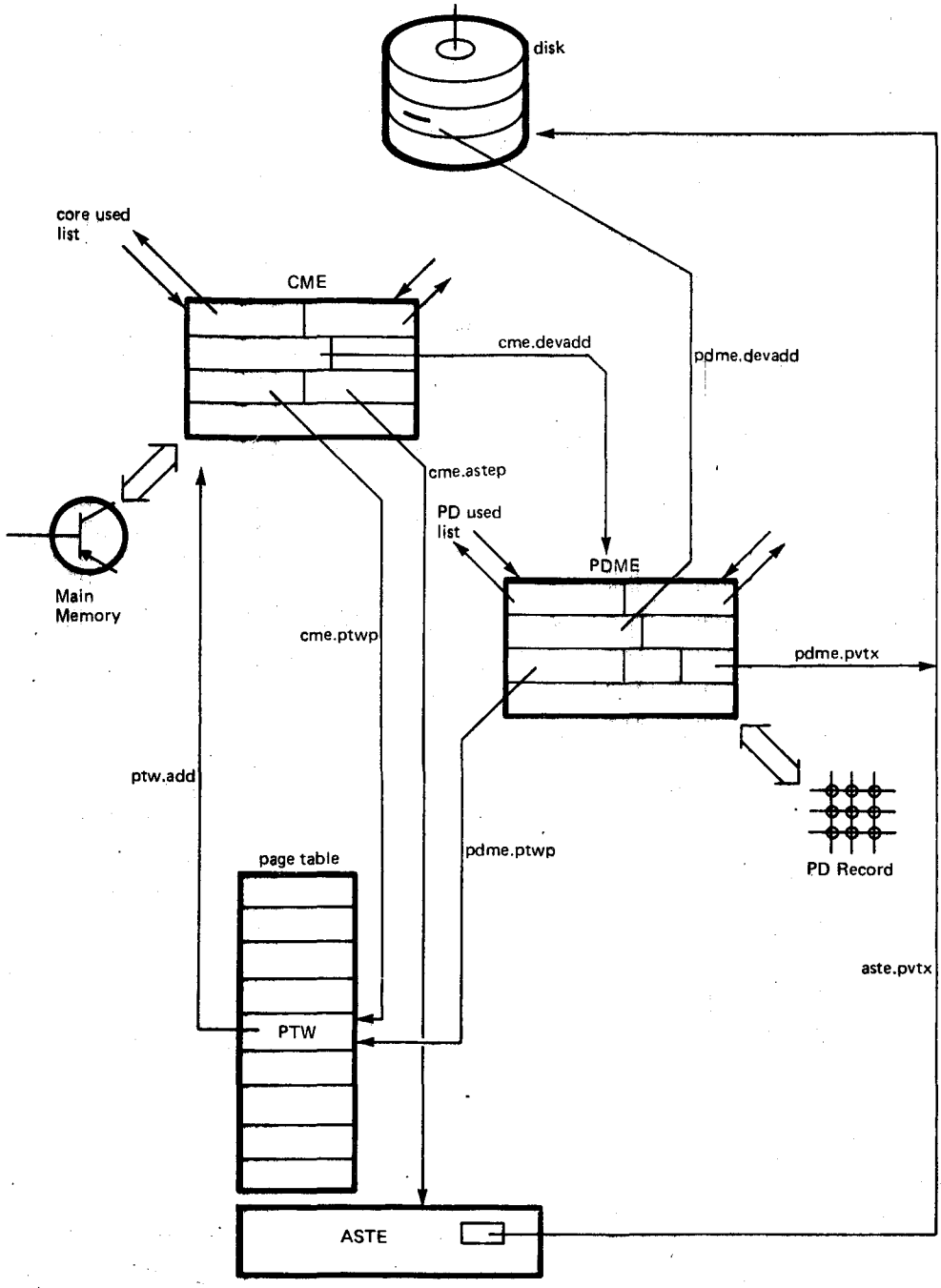


Figure 6-3. Page Control Data Bases
Page in main memory and on paging device

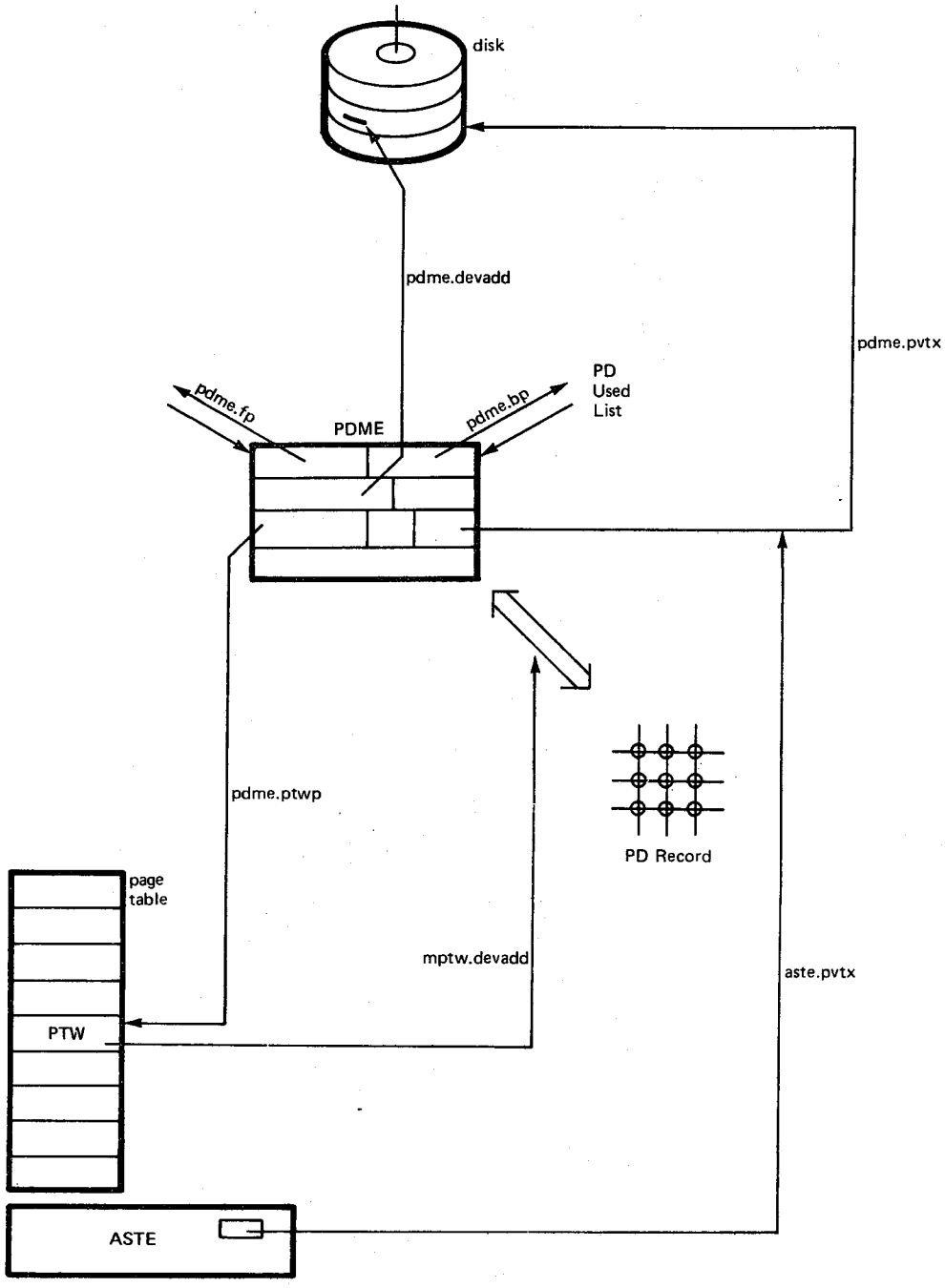


Figure 6-4. Page Control Data Bases
Page on paging device, not in main memory

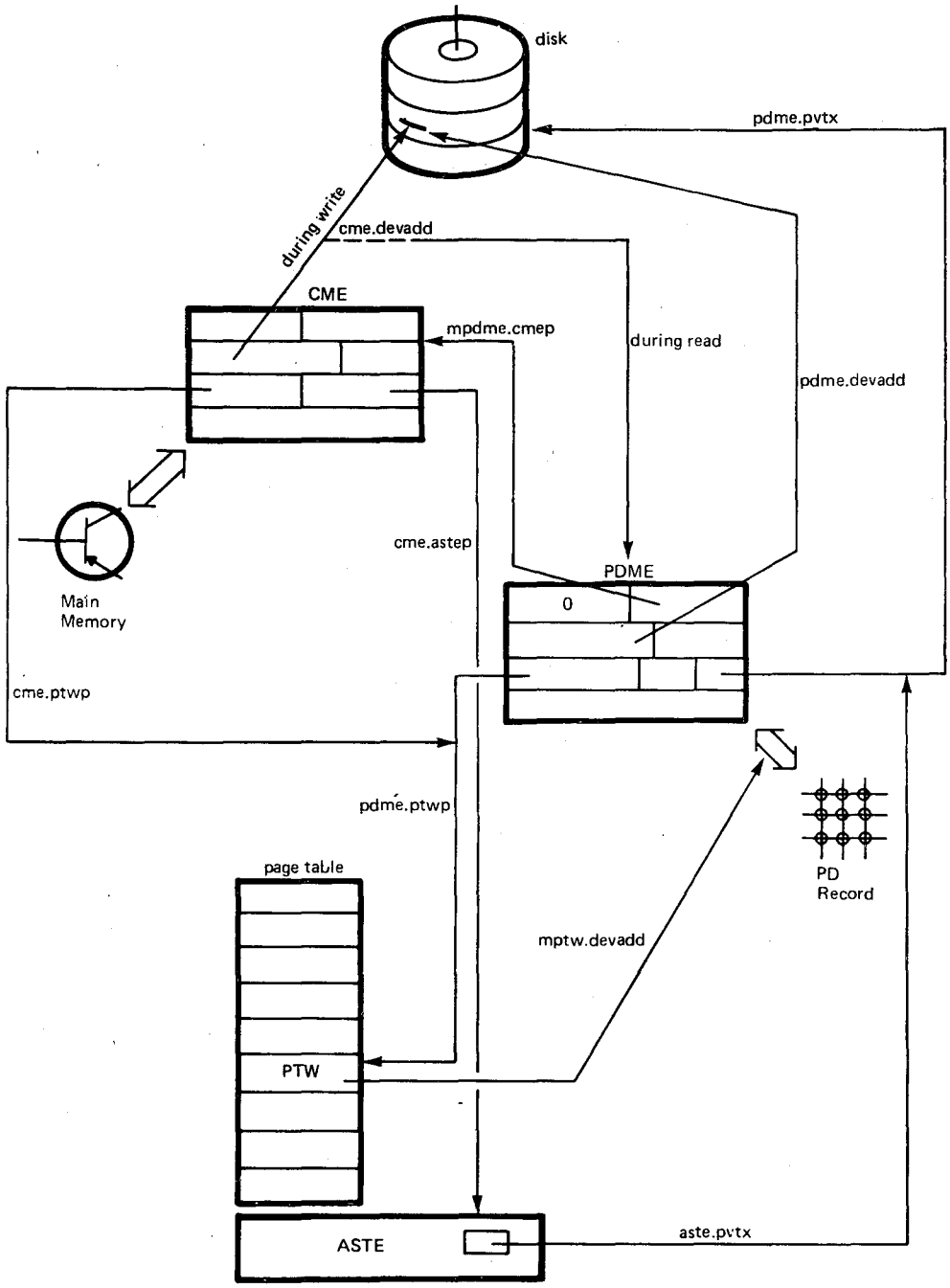


Figure 6-5. Page Control Data Bases: Read-Write Sequence

SECTION VII

ADDRESS MANAGEMENT POLICY

INTRODUCTION AND NULLED ADDRESS

The address management policy of Multics is that set of designs and their implementations which manage when record addresses are assigned to pages, the state of the relationship between the contents of each page and the contents of any secondary storage record which may be assigned to it, and the deassignment of secondary storage addresses from pages.

Some address management policy must exist, as this service is a necessary one of page control, a service to its own internal workings. The goals of the Multics address management policies are these:

1. No record address shall ever appear in a VTOCE unless it is known with certainty at the time it is put there that the data in the associated disk record is the data from the page of the segment which has that address as its record address.

2. No record address shall ever be made available, by placing it in the free pool of records on its physical volume, until it is known with certainty at the time it is so made available, that it has been purged from the VTOCE on disk in which it resided.

3. The observance of points 1 and 2 can be shown to imply point 3, to wit, no record address shall ever appear in more than one VTOCE of a given physical volume at the same time, not even during any transitory or inconsistent states. Such states shall not be allowed to exist.

4. No page of data will be allowed to be created unless a disk record is available to be assigned to it at the time it is created (by being faulted in).

5. The supervisor, when running in any process, shall never encounter a condition where a supervisor data base, stack, or procedure, cannot be grown because of lack of space on its physical volume.

6. The system must be capable of being bootloaded without any knowledge of which addresses are available for assignment. These maps can only be constructed by running software to construct them. This software consists of paged segments, and these segments must reside somewhere.

7. The system shall not deplete its available space on any volume simply as a result of being bootloaded, i.e., shut down and brought up repetitively, or just running an extended or arbitrary period of time.

The address management policy takes cognizance of the fact that the system can crash at any time. A total power failure can cause this. When the system has crashed in such a way that the contents of main memory are lost, or in general, emergency shutdown does not succeed, the next bootload must make the best of what is in the storage system hierarchy as it encounters it. Thus, it is one of the highest goals of address management to make sure the the instantaneous state of secondary storage, at any instant, is never such that the next bootload will give away data by accident or place data in the wrong place.

To understand this more fully, an example must be given of address management policy failure in the pre-4.0 storage system. The following scenario is impossible under the current storage system.

1. Segment A contains a PL/I program. Its owner deletes it, freeing its record addresses, but leaving the data in those pages. The directory file map (predecessor of the VTOCE) is freed.

2. Segment B gets created. Someone types a sensitive letter into it. A record of disk gets allocated for a page of this segment, and is written out. It is a page that used to belong to segment A.

3. The directory page which had A's branch has not yet been written out, as this directory is heavily used, and thus not evicted from main memory.

4. The page of the personal letter gets written out.

5. The system crashes unrecoverably.

6. The next bootload finds segment A still there, as the page of the directory containing the branch never got out to disk. What is worse, one page of this PL/I program now contains a page of the personal letter.

This situation is known as a reused address; due to asynchrony in the updating of pages to disk, two segments claim the same record address. What is worse, the data from the new one is in the page that is described by the file map of the old one. It is the principal goal of the release 4.0 and later address management policy to categorically avoid this and a whole class of similar problems.

It can be seen that if points 1, 2, and 3 above are followed rigorously, the scenario above can never happen. These rules serialize the deallocation and reallocation of addresses so that any trace of any given record is completely gone from one segment before it is freed, and thus made available for use in any other segment.

Point 1 specifically, makes it necessary to make finer distinctions between the states of "there is no disk address associated with a page" and "there is a disk address associated with a page". These finer distinctions did not exist in pre-4.0 versions of the storage system. Consider the case of a page of a segment that has never been written to disk. Now surely, one must allocate a record and associate it logically with that page before writing it, so there must be a finite time between those two operations. There is also the entire time during which the request to write is in the disk DIM queues, when it has not yet been written. Consider the case of a request to "Update the VTOCE" of the segment during this time. Should the address be reported to the VTOCE or not? If it is, and the system crashes before the page gets out, then an address appears in a VTOCE which denotes a record of disk with the left-over residue of some other segment, a security problem. If not, then some finer distinction must be made about the nature of assignment to tell when to update addresses and when not.

This is precisely where the concept of the nulled, or semi-killed device address enters. Point 4 above implies the association of record addresses with pages at the time that null pages are faulted into main memory. A null page is one that is in no way associated with any record of disk, and whose contents are logically zero. The association of this disk record with the page is now in that state given in the precious paragraph, where it is known that it does not contain data from that segment, and may not be reported to segment control. An address in this state is called a nulled or semi-killed address. It is a disk address. It is assigned to a page, but the contents of the page are zero, and the contents of the disk record are residue from some other segment, the nulledness of a nulled address is encoded intrinsically in its representation.

The opposite of a nulled address is a live address. A live address may be reported to the VTOCE, via pc$get_file_map, at any time. Its state of being live implies that that record of disk is known to contained data from the page of the segment which has this live disk address as its disk record address.

The act of converting a nulled address into a live address is called resurrection. Since an address being live means that it is known that a given page has been written there, resurrection happens at the successful completion of any of various disk-writing operations, namely:

1. Any page write from main memory to disk.
2. A read-write sequence (RWS) from paging device to disk.
3. A double-write, when the paging device is being used in write-through mode (see sst.double_write in Section VI).
4. A post-crash repatriation RWS. (See Section IX, "Post Crash PD Flush").

Live addresses can also be dynamically nulled, converting them into nulled addresses. This happens in two cases:

1. When the page is destroyed, via truncate, which includes all cases of segment deletion.
2. When the page is discovered to contain zeros (See "Zero Pages" in Section V.)

When a live address is so nulled, again, zeros become logically associated with the page, and the address is not reportable to a file map. In this case, the page of disk contains a residue again, in specific, the residue of an older version of that page of that segment.

The force of the above policies is that addresses in a VTOCE, as described in the introductory sections of this manual, have only two possible meanings:

1. A Null address: This page of this segment logically contains zeros.
2. A Record address: This page of the segment is contained in the disk record designated.

Therefore, at the time that a VTOCE is updated, the many fine divisions of state of the page and its address must be mapped into one of these two states for the file map being updated, depending on what action is intended for the next bootload should the system crash irrecoverably the next instant. Thus, all states involving nulled addresses are reported to the VTOCE as in case 1 above, via the reporting of a null address to the file map. Now the reporting of a null address to a VTOCE where perhaps previously there had been a live address, is the sole precondition, acceptable to point 2 at the beginning of this section, for depositing (freeing) a record. Thus, at the time that a file map is reported to segment control, a list called the deposit list is also reported: it consists of all of the nulled addresses found in the segment, for pages which were not in main memory or on the paging device (in these cases, it would violate point 4 to deposit their addresses). Page control's association between the page and the disk record is broken at this time by placing a null address in the PTW devadd field and reporting it to the file map, the logical contents of the page remain zero, but no page of disk is associated with the segment.

Segment control holds on to this deposit list. It updates the VTOCE, causing the addresses being deposited to be replaced by the null address gotten above. When and only when this VTOCE write has been determined to be successfully completed, are these addresses (the deposit list) handed in to pc$deposit_list to actually be marked as usable by some other segment. The special entry in the VTOC manager, vtoc_man$await_vtoce, exists solely for the purpose of waiting for successful completion of VTOCE I/O for this reason. The same action is taken when freeing a VTOCE is used as a means of invalidating its contents, when addresses are involved. This is also done by the segment mover. See the descriptions of "VTOCE Updating" and "Segment Truncation" for the impact of these policies on segment control.

IMPLICATIONS OF FINITE PACKS

Each disk pack in the current technology has a finite capacity on the order of tens of thousands of Multics records. Each device address used by page control and segment control is relative to some particular pack: thus the size of these various fields limits, and is limited by, the amount of storage available on one pack.

Each segment resides on one and only one pack: this fact is intrinsic to the interpretation of the device addresses designating records on that pack, as they are only meaningful with respect to a pack designated by the PVT index in the ASTE of the segment in whose data bases they are found. (Note, however, that segments can and do migrate automatically between packs: See Section II).

Since all pages of all segments are assumed to be zero until otherwise known, record addresses are not actually assigned until pages are actually used. In older versions of the storage system, address assignment happened when a page was first evicted from main memory, and was found not to be zero. Since all addresses were withdrawn from the same single large pool, this operation could only fail if the entire system were out of disk, i.e., there was not one more record available anywhere. However, since each pack now has its own pool of free storage, the case of a segment not being able to be evicted because there is no place to write it is a serious one. Such a page would tend to become "stuck" in main memory until some (presumedly complex) action would be taken to recover. An arbitrary number of such pages would tie up an arbitrary amount of main memory. What is more, if the system chose to take a brute-force approach to evict the page, it would have to destroy the user's data, with no particular reason or even good method of telling him or her.

Thus, point 4 above is made. No page of data is allowed to be created (implicitly always as zeros) in main memory, which is the only place pages get created, unless a record is available at that time for assignment. Since it will probably have to be written out later, it is better to find out now if no disk is available. The unsatisfied page fault can be used to make the entire segment-moving mechanism handle the problem transparently if this is done. What is more, the nulled address concept precisely expresses the relation between the page of the segment and the record address so assigned at this time. This unsatisfied page fault is also critical to the implementation of the mechanism that allows page faults on the FSDCT to be simulated by the page fault handler.

It is, of course, always possible that the user process might only reference that page, or never store anything into it but zeros. We cannot rely on that. There is a potential here for interaction with access control to ensure this, but this is not exploited at the current time.

NON SEGMENT-MOVABILITY OF THE SUPERVISOR

The supervisor may not run out of physical volume space at any time. That is to say, if it is necessary to create a page of the supervisor's stack, and there is not a single record available on the volume on which it resides, the system is in an unrecoverable situation. Any software which did any action at all would have to run on that stack, and it cannot be used. Thus, all supervisor data bases, in particular, the ring 0 stack (PDS) of each process, must be assigned addresses at the time it is created as a normal segment, before it is used as a ring-0 stack. This implies a cooperation of page control and segment control. (See "PDS and KST Management", in "Services of Segment Control" in Section IV). Addresses are assigned to the PDS of the process being created by touching every page of it. This causes nulled addresses to be assigned. However, since this segment is part of the storage system hierarchy, the periodic VTOCE update of the AST Trickle (See "AST Trickle" in Section II) would tend to deposit these addresses, as the above paragraphs have stated is the fate of nulled addresses at VTOCE update time. In order to suppress this depositing, the AST bit aste.dnzp, which normally suppresses nulling of the addresses, of zero pages, or checking for them, is viewed in conjunction with the bit aste.ehs, the "entry hold switch" making these ASTE's semi-permanently activated, by pc$get_file_map, to suppress reporting and making-null of these nulled addresses.

This action of pre-assigning addresses is called prewithdrawing. All of the supervisor data bases, such as the stack used at shutdown time, the FSDCT, the dirlockt_seg, the lock segment, etc., are all prewithdrawn at the time they are created by Initialization so that the supervisor does not run out of disk in an embarrassing place. There is another reason for prewithdrawing these segments at the time that they are created: it is a consequence of points 6 and 7, which are now discussed.

GUARANTEED BOOTABILITY OF THE SUPERVISOR

The segments that compose the hardcore supervisor, including all data bases, and all parts of all salvagers, must, if paged, have disk addresses assigned. By virtue of the policies given above, these pages, as all other pages managed by page control, must have addresses assigned at the time that they are created.

If the system has crashed without a successful ESD, then the volume map of any volume present during that bootload will not be valid. (The volume map is the disk copy of the FSDCT bit map for that volume, copied into the FSDCT when the volume is accepted and written out when demounted). The supervisor must have some place to allocate its own pages during the next bootload. Since no volume map may be believed, the supervisor must in effect be booted on a volume not present during the last bootload.

Rather than inflict this difficult operational restriction, a "special volume" called the hardcore partition is defined on the root physical volume (RPV) of a given hierarchy. In effect, every time the system is booted, the supervisor is booted "cold" into the pseudo-volume of the hardcore partition. This is to say that the volume map of the hardcore partition is defined to be entirely full of "free" markings for its pages. Therefore, the supervisor may construct the FSDCT bit-map for the hardcore partition out of "ones" for the length of the hardcore partition. The supervisor may thus allocate pages anywhere in the hardcore partition. (Since the bit-map is wholly fabricated, there is in fact no volume map on disk for this region). The location and extent of the hardcore partition are stated in the volume label of the RPV, and are not subject to change during running of Multics (See Section XIV).

It is a corollary of the definition of the hardcore partition as a region totally free upon bootload that all of the contents of pages in that region, of that bootload, will be undefined (as the records are being reused) during the next bootload. Now only two classes of segments will have pages in the hardcore partition: supervisor segments (without branches or VTOCEs) of that bootload, and deciduous segments (essentially supervisor segments with branches and VTOCEs). The non-deciduous supervisor segment will not be accessible during a subsequent bootload; all information about them was contained in their ASTEs, and is gone. The resources consumed by them in the hardcore partition are reused by virtue of the above definition. The deciduous segments, on the other hand, will have pages all over them being reused by new segments. Therefore, deciduous segments can not be used from one bootload to the next; an attempt to activate a deciduous segment of a previous bootload causes a connection failure. When deciduous segments are deleted, by the next bootload, their pages are not deposited; the records in the hardcore partition are reused by the current bootload by virtue of the definition of the hardcore partition.

All supervisor segments, deciduous and otherwise, are totally prewithdrawn against the hardcore partition with very few exceptions- see below). This means that a given hardcore partition must be capable of holding the supervisor in its entirety, or the system will crash with an out-of-physical-volume condition during initialization. Thus, deciduous segments' record addresses are totally in the hardcore partition, and all of their pages become invalid during the next bootload. This property has been likened to the perennial defloration of flora: that is why deciduous segments are so called.

The bit-map of the hardcore partition is used as the only free storage map for the root physical volume, onto which the system is booted, until the middle of collection 2, when the program accept_fs_disk$rpv runs (See Section XIV). If the system crashed in the prior bootload, the physical volume salvager will have been invoked before this point in the bootload to reconstruct the volume map of the RPV, in addition to other functions. Thus, at this point in the bootload, the real volume map of the RPV replaces the map constructed for the hardcore partition. (No addresses in the hardcore partition should ever be deposited after this point). Thus, all requests for new record addresses on the RPV, will cause records to be withdrawn from the real volume map of the RPV.

The fact that the real volume map of the RPV replaces that of the hardcore partition means that any page withdrawn against that map by the supervisor must ultimately be deposited, or the system will run out of disk on the RPV by virtue of continued operation, a situation explicitly disallowed by point 7 at the beginning of this section. Thus, if supervisor data bases grow, i.e., acquire disk records, after the point mentioned above in initialization (the "acceptance of the RPV volume map", the supervisor must, in order to perform a successful shutdown, truncate these data bases and deposit these addresses to keep point 7 true. Not only is this difficult because of the need to differentiate the hardcore-partition addresses from the ones withdrawn against the real RPV volume map, but this systematic self-destruction of the supervisor causes any problem in shutdown to be hard to diagnose, as the supervisor has willfully partly destroyed itself at that time. It is also difficult to organize a supervisor shutdown which proceeds by destroying itself. (In fact, pre-4.0 versions of the supervisor destroyed themself in just this way, and continually had problems in locating every last record that had to be deposited, and doing it in the right order). Thus, the entire supervisor, with the exceptions noted below, is prewithdrawn against the hardcore partition at the time it is created, for this second reason.

There exists a small set of segments, called "delete_at_shutdown" segments that are managed in complete violation of points 5 and 7. These segments are part of the supervisor. They are data segments that are:

1. Large, and may not even be used for their full length.

2. Non-critical were the supervisor to run out of disk on the RPV were these segments to encounter an OOPV condition.

These segments are managed this way simply to avoid having to make the hardcore partition large enough (an issue of a few hundred records) to contain them were they prewithdrawn against it. Thus, these segments are truncated during a successful shutdown, contain both hardcore-partition and real-RPV-volume map addresses, and may encounter out-of-disk conditions.

The bit slte.delete_at_shutdown, set from the MST generator "delete_at_shutdown" keyword makes a segment so. Such segments are kept in the "hardcore" ASTE list, to facilitate the truncation at shutdown time.

RPV PARASITE SEGMENTS

There are some segments, such as the descriptor segments of all processes except the initializer, and the PRDS of all processors other than the Bootload Processor, which reside on the RPV, but do not have VTOCEs or branches. Thus, page creations for these segments withdraw against the real RPV volume map. In the case of a normal shutdown, orderly process destruction and deconfiguration frees these pages, assuring that the system does not run out of disk by virtue of continued operation (point 7). However, in the case of a crash, with or without a successful emergency shutdown, these orderly destructions do not occur, as all of the relevant processes may be in inconsistent states. Since these "RPV parasite" segments have no VTOCEs, the deletion of process directories performed by system answering service startup does not free their pages. Thus, a volume salvage of the root physical volume (so-called "short RPVS") is performed automatically after every crash. This salvage collects all space not described by VTOCEs, making it available for reuse. This includes all space used by RPV parasite segments.

abs-segs (EXPLICIT ADDRESS MANAGEMENT)

Many "segments" in the supervisor are not segments at all, but rather segment numbers, and possible ASTE/page tables, used for addressing main memory, bulk store, or disk. Such "segments" are known as abs_segs. There are two "levels" of abs-seg, the SDW-level abs-seg and the PTW-level abs-seg. An SDW-level abs-seg is used by placing an SDW describing a region of main memory (as a segment) in a position in the descriptor segment, or an SDW describing a page table (as the page table for a segment). The extent of main memory, or the segment described by the page-table "become" the "segment" whose segment number was that of the position in the descriptor segment into which the SDW was placed.

For a PTW-level abs-seg, the SDW always describes the same page table. The PTWs of this page table are filled in with the disk addresses of a region of disk or bulk store (the PVT index of that drive or the bulk store (see sst.bulk_pvtx in Section VI) is placed in the field aste.pvtx), and all references to that segment "become" references to that extent of disk or bulk store, i.e., the segment number's segment "becomes" that region of disk or bulk store.

If this reminds the astute reader of the method used to access every single segment in the Multics storage system hierarchy, that is because indeed it is. The difference is solely one in orientation. For an abs-seg, the segmentation and paging mechanism, and the implicit services of page control, are being used as a technique to read and/or write disk. For a hierarchy segment, segmentation and paging and the implicit services of page control and segment control are used to make a collection of disk records "behave" like a segment. There is no physical difference to the two techniques.

SECTION VIII

MECHANISMS

The mechanisms of page control are those policies, protocols, and programs that compose the internal organization, and support the services thereof. This section details those policies, protocols, and programs. Some policies, such as the address management policy, and the main memory and paging device replacement algorithms, are not manifestations of internal organization, but rather artifacts of the services page control is called upon to perform. Such policies have already been explained.

Those policies already described are the externally visible policies. Some of them have become documented in the literature, and thus acquired some measure of fame. Yet it is the policies and mechanisms explained in this section that are little-known, but necessary to the debugging of problems, interpretation of crash dumps, and contemplations of functional or organizational improvements to the whole of page control

The section is divided into three parts:

1. Policies, protocols, and organizations.
2. Individual mechanisms.
3. Internal interfaces.

The first part describes strategies and principles in effect throughout page control, and critical to its external interface. The second describes particular mechanisms, that are ostensibly divorced from the explicit services, such as the method of waiting for page faults, the "recursive" FSDCT paging, etc. The third part describes interfaces that are in effect the services of page control for page control, such as most of the entries to the transfer-vector "page."

POLICIES, PROTOCOLS, AND ORGANIZATIONS

Global Page Lock

All manipulations of page control data bases, with the exceptions noted below, must be performed under the protection of the global page table lock. No process that has the global lock locked may give away or accidentally lose the processor on which it runs. Thus, any process that has the global lock locked must be masked to "sys_level", and have its stack, linkages, and procedures wired, not referencing any non-wired parameters, code, or data bases.

There is no general mechanism for multiprogram-waiting on the page-table lock. Except for processes taking page faults, all attempts to lock the page table lock are performed by looping on it. Internal to ALM page control, this is performed by executing:

```
    tsx7     <page_fault>|[lock_ptl]
or  tsx7     <page_fault>|[lock_ptl_no_lp]
```

depending on whether or not the caller has set up a stack frame. This procedure may be generally accessed as page$lock_ptl from PL/I code, yet this is rarely done (only the loading function, wired_plm, does this), as all other PL/I procedures that lock the global lock also wish to wire their stack frames and mask to sys_level; this compound function, which includes calling page$lock_ptl, is performed by the very common call:

```
call pmut$lock_ptl (save_mask, save_ptp);
```

The two parameters are used in the corresponding unlock call:

```
call pmut$unlock_ptl (save_mask, save_ptp)
```

to identify the PTWs wired by the first call, and the old mask. This mask variable has the old wired bits of the PTWs embedded in it, and is intended for use only by pmut$unlock_ptl.

There exist calls to unlock the page table lock, these involve interaction with the traffic controller in order to support the page table lock multiprogramming feature described in the second part of this section. This call is:

```
tsx7     <page_fault>|[unlock_ptl]
```

in ALM page control, with the transfer vector page$unlock_ptl and pmut$unlock_ptl having the same relation as the corresponding lock entries (pmut, however, does not use page$unlock_ptl, but rather page_fault$pmut_unlock_ptl, a side door to the unlock mechanism which avoids pushing extra stack frames).

The page-fault handler, the fault side of page control, has a mechanism for waiting, via the traffic controller, for the page table lock to unlock. The lock_ptl routine in page_fault takes special action when invoked by the fault side; this mechanism is explained in the second part of this section.

There are two large classes of page control manipulations that may be performed without having the global lock locked:

1. The turning on/off of wired bits of the PTWs of supervisor or semi-permanently activated segments.

2. The construction or destruction of the page tables of inaccessible segments.

In the first case, the bit ptw.wired, used by the main memory replacement algorithm to avoid eviction of a page, may be turned on or off at any time by any process that is keeping track of what it is doing. Page control, operating under the page-table lock, never turns wired bits on or off except in two cases:

1. Loading of processes' critical pages.

2. Abs-wiring of I/O buffers

Thus, processes may turn on "wired" bits of PTWs for segments such as the ring-zero stack (pmut$lock_ptl does just this) without fear that page control might be trying to turn them off. The restrictions on this type of activity is that one must choose the segment with care: its AST entry must not be removable, lest these PTWs vanish while being dealt with, or before having their wired bits turned off. Thus, only supervisor segments and semi-permanently activated segments (including PDSs of other processes than the initializer) are eligible for such treatment. Furthermore, this mechanism is not shareable; unless some external means is used to organize such wiring requests (such as wire_proc, see Section X, or the I/O Buffer Manager iobm, only segments known to be essentially unshared may be so dealt with (limiting this almost exclusively to ring-zero stacks (PDSs). Once wired bits are so turned on, simply touching the page whose PTW was manipulated, bringing it into main memory, will "wire" it, since it now may not be evicted.

Unwiring of pages so wired may be done by simply turning off the wired bits; it was guaranteed by the preconditions of the last paragraph that the PTWs cannot have disappeared, and no other process could have turned off the wired bits, or worse yet, wanted them kept on. This is the method used to "unload" processes, i.e., unwire their critical pages, without the protection of the page table lock. In fact, an extension of this mechanism is used by the I/O buffer manager to turn off the "abs_wired" bit (cme.abs_w) in the core map entry without the protection of the lock, for the definition of abs-wiring is that the page, and hence, the core map entry it is associated with, may not be moved.

The other broad class of manipulations performable without the page table lock locked is that concerning itself with segments that are inaccessible. A segment being activated by definition has no SDWs describing it, and has no pages in main memory or on the paging device. Thus, any manipulations on its PTWs or AST entry can have no effect on any of the data bases of page control, since no CMEs or PDMEs describe these PTWs or ASTE. A segment that has been "finalized" by pc$cleanup (see "Services," Section IX) again has no pages in main memory or on the paging device; since making the segment inaccessible is a precondition for calling pc$cleanup, such a segment is in the same state, and its PTWs may be dealt with as fitting.

There are two smaller classes of manipulations performable without the page table lock being locked:

1. The validation of page control events by the traffic controller.

2. The depositing of addresses.

The traffic controller interacts in a close fashion with page control to perform Process Loading (see "Process Loading" in "Services"). Among the quantities returned by page control to the traffic controller, when this service is performed, is a wait event. The validity of this wait event is verified under the traffic control lock by the traffic controller, under whose lock all notifications must be performed. This validation is performed by checking out-of-service bits, the particular location of which may be inferred from the value of the "wait event" (see "Wait Protocols" below). If these bits are not on, it is a certainty that the event in question has already happened; if it had not, these bits would still be on, regardless of any lock anywhere, and the

traffic controller effectively proceeds with the loading operation, which is, in effect a conservative action for the traffic controller. (The worst possible result of such a mistake would be to retry the loading an extra time.) On the other hand, if the bits are on, the traffic controller assumes that the event has not happened. This is not fully correct; it may have happened already, and a new similar event started. If any such event is in progress, a "notify" will be forthcoming if and only if the "notify requested" bit in an appropriate PDME or CME is on. In the case of the legitimate event being waited for, it always is. In this peculiar case above, it may or may not be. The traffic controller assumes, if the out-of-service (or RWS, as appropriate) bits are on, that a notify will be forthcoming, and sets the process being loaded waiting on that event. The worst possible outcome of a mistake (highly unlikely) in this decision would be a 90-second "notify timeout," and retry.

The depositing of addresses, i.e., the marking of bits in FSDCT bit-maps as free is performed outside of the page table lock. Withdrawing is performed under the protection of the page table lock. The latter is necessary, as were there no lock protecting this withdrawing, two processes might "succeed" in withdrawing the same address simultaneously, resulting in not only a "reused address," but an inconsistent FSDCT and PVT. Thus, withdrawing is performed under the lock. Depositing need not be, because no two processes can be trying to deposit the same address at the same time, because there are no reused addresses in the system. Each address appears at most in one place at one time. Furthermore, no process is specifically trying to withdraw any given address. Depositing consists of turning on a bit and incrementing the free-record count, both of which operations can be done without the protection of a lock. If the address being freed was already free ("unprotected address," a cause for crash) it will be free whether or not the lock is locked. If it is not, no other process is trying to free it. One implication of the fact that depositing is not performed under the page table lock is that the depositing procedure (free_store, called only by pc) takes page faults in the normal fashion on the paged, non-wired FSDCT, while other processes are so doing and the "recursive" page fault simulator is accomplishing "withdraws" on perhaps the same pages.

The page table lock is lower in the locking hierarchy than the traffic controller lock. It is lower than any of the locks used by the storage system DIMs to control their data bases, and thus lower than any locks used by the IOM manager.

It is higher than the lock used by the I/O buffer manager, and thus higher than any locks used by the I/O interfacer.

It is a "wired" (per-processor) lock, and thus higher than any non-wired (per-process) lock, such as all directory locks and the AST lock.

Wait Events Used by Page Control

Page control uses two "waiting" type mechanisms:

1. Looping and retrying until some asynchronous event happens; used to wait for the completion of bulk store I/O, the clearing of the page table lock (by other than the fault side), or the dying-down of disk queue traffic ("running the disk DIM").

2. The wait/notify mechanism of the traffic controller.

The first method is used where giving away the processor is impractical or impossible, including several "worst-case" type situations. The wait/notify mechanism of the traffic controller is used to wait for precisely three types of events:

1. The completion of any disk paging I/O, i.e., disk read or writes of pages to and from main memory for any other reason than a read/write sequence (RWS).

2. The completion of read-write sequences (RWSs).

3. The unlocking of the global page table lock, awaited only by the fault side.

There is also the temp-wiring table used by wire_proc, among the peripheral services of page control, but it is far removed from the internal organization of anything else in page control. (See Section X for more on this.)

Each event for which page control waits has a 36-bit "Event ID," as must be true of all events waited for via the traffic controller. Part of the protocols of using the traffic controller wait/notify mechanism is that event IDs need not be unique over the system, and thus notifies can occur spuriously as event IDs clash. However, event IDs generated by page control are unique within page control. Page control, when looking at an event ID it generated can determine with certainty what event is associated with that event ID, and whether or not it has happened. There are three classes of event IDs corresponding to the three types of events above:

1. A binary number in the right-hand half of a word, whose left half is zeros, this number being bigger than the offset in the SST of the first ASTE (the word offset of the pointer sst.astap), is the offset of a PTW in the SST. Such an event ID is associated with the event of the completion of non-RWS disk I/O for that page.

2. A binary number in the right-hand half of a word, whose left half is zeros, smaller than the offset in the SST of the first ASTE (the word portion of the pointer sst.astap), is the offset of a paging device map entry (PDME). Such an event ID is associated with the event of the completion of an RWS for that PD record.

3. The octal constant "160164153152"b3, being the ASCII for "ptlk", is associated with the event of the unlocking of the global page table lock.

A "PTW event" (Case 1) may be tested for having completed by the being-on of the bit ptw.os. A "RWS event" (Case 2) may be tested for completion by the being-on of the bit pdme.rws in the PDME designated by the numerical value of the event ID. These checks must be made under the page table lock, via an organized methodology explained below ("Wait Protocols"). The "PTL event" (Case 3) may be tested for having completed by inspecting the contents of the page table lock, sst.ptl.

PTW events are also used to express the event associated with the completion of non-RWS bulk store I/O. However, these events never leave page control and thus are never waited for via the traffic controller. Page control "waits" for PTW events corresponding to bulk store I/Os by means of calling the bulk store DIM "run" entry until the event has occurred.

Wait Protocols of Page Control

Part 1 - Waiting for a given single event - other than the PTL event (Simplex Wait Protocol)

The methodology used in page control to wait for an event is strongly dependent on which side of page control is doing the waiting. For a start, the interrupt side never waits, or has to wait, for any event (unless loop-locking the global lock is considered waiting for an event). Thus, the interrupt side may not run the replacement algorithm, which would "wait" for disk I/O to die down by looping.

One must consider the code of the process-loading function a separate "side" of page control here; it is the only function that acts on behalf of some given process, including causing that process to wait, but is never actually called by that process.

The page control wait mechanism is not used so that page control may wait; rather, it is used so that processes on behalf of whom page control is performing services may be made to wait, when awaiting page control events is necessary to the fulfilling of that service. This is to say, that when the main memory or paging device replacement algorithms start a write or RWS respectively, page control has no need, in general, to wait for its completion. On the other hand, some process that is trying to drive all pages of a segment out of main memory and paging device may well have to wait for the completion of such a write or RWS, whether it had started it or it had already been in progress. Similarly, a process taking a page fault must be made to wait for a disk I/O completion if a disk read was involved in resolving that page fault. Thus, the procedures that implement the services of page control may often have to wait for I/O completions in order to carry out these services as specified; the mechanisms of page control never wait.

The completion of all page control events is detected and determined by page control. No external agencies in the system wait upon or notify page control events. What is more, the "notify" operation for all page control events is performed under the page table lock, usually by the interrupt side of page control. The occurrence of a PTW event consists of the turning off of the PTW out-of-service (I/O in progress) bit. The occurrence of an RWS event consists of the turning off of the PDME RWS (pdme.rws) bit. These events can only happen under the page table lock. Page control does not perform a traffic-control notify every time a PTW event or RWS event occurs. PTW events are notified only if the bit cme.notify requested in the CME of the main memory frame in which the I/O was taking place is on. These notify operations take place in the traffic controller, but under the page table lock. These notify-requested bits are turned on when and only when page control has made the decision that a process must wait for such an event, at such time, the associated notify_requested bit will be turned on (all under the page table lock).

The decision to make a process wait happens in three different ways, depending on whether the decision is performed by the fault side, the call side (other than the loading function), or the loading function. In the first two cases, the process executing the code will be the one that waits; in the third case it will not.

The fault side makes the decision to wait at the end of page fault processing, all under the page table lock. The readin of the page faulted on, if nonnull has already been initiated (see "Services" Section IX, "Page Fault Handling"). The PTW of the page faulted on is inspected. If the PTW indicates that the page has already been read in (or created, in the case of zero pages), the page fault machine conditions, and thus the faulting Control Unit cycle, and thus the instruction, and the program that took the page fault, are restarted (after unlocking the page table lock). If, on the other hand, the PTW indicates that the page has not been (completely) read in, there is waiting to be done. Since this process has the page table lock locked, and notices that the page is not in, it does not matter whether or not the page has actually come in, i.e., the disk data transfer has been performed. The interrupt side, which is the only agency that can turn off that bit ptw.os or pdme.rws, (cause the PTW or RWS event to occur), cannot be invoked until this process releases the page table lock, or itself invokes the interrupt side under the page table lock. In the case where there is waiting to be done, the subroutine read_page, invoked by the page-fault handler, has returned the event ID of the event that must be waited for. If the page being read in is undergoing an RWS, this is an RWS event. Otherwise, it is a PTW event. If the page requires an allocation of a record, and the appropriate page of the FSDCT is not in main memory, it may be an RWS event or a PTW event for a page of the FSDCT (see "FSDCT paging" later on).

The fault side waits for the event so given to it by read_page in the following way:

If this event is an RWS event, identify the PDME designated by the RWS event, and turn on the abort bit. This causes an RWS abort and a notify of that RWS event at the time the RWS completes. A branch is executed (pxss$page_wait in the traffic controller) to wait for that event and unlock the global lock.

If this event is a PTW event, determine whether it is for a bulk store transfer or a disk transfer. If the devadd in the CME for the page frame denoted by the PTW is a "paging device devadd," it is a bulk store transfer. Otherwise, it is a disk transfer unless the segment is the "pdmap_seg," abs-seg, an abs-seg used to read the bulk store as though it were a disk. Then it is a bulk store transfer. If it is a disk transfer, turn on cme.notify_requested in that CME, and go to pxss$page_wait to wait for the PTW event. This bit will cause a notify of that PTW event when the I/O completes. If this is a bulk store transfer, call the "run" entry of the bulk store DIM, and check whether or not the PTW out-of-service bit has gone off and call the "run" entry of the bulk store DIM in a loop, until this bit has gone off. The "run" entry of the bulk store DIM will interrogate the hardware status of the bulk store, and call the interrupt side of page control, potentially causing the PTW event to occur, as its function. Then restart the machine conditions.

Bulk store transfers are not awaited via the traffic control mechanism because the transfer time of the bulk store is comparable to the overhead time spent going through the traffic controller.

Thus, a process taking a page fault either restarts the machine conditions at the end of a page fault, or goes to the traffic controller to wait for either an RWS event or a PTW event corresponding to a disk I/O. In either case of going to the traffic controller to wait, a bit will have been turned on (pdme.abort or cme.notify_requested) which tells the interrupt side to notify via the traffic controller.

When a process waits on behalf of the fault side of page control, (this includes waiting for the lock, see "Traffic Controller Interface" below) no other information is recorded about the state of that process other than the machine conditions from the page fault that was taken, and the fact that it is indeed waiting on behalf of the fault side of page control. When that event is notified, the traffic controller branches to page_fault$wait_return, which does not lock the page table lock, modify, or even inspect page control data bases in any way, but only restarts the machine conditions of the fault. If indeed the PTW was made to describe main memory as the interrupt side noticed an I/O completion, and the page has not been evicted in the interim, the interrupted machine cycle will be retried and completed. If not, another page fault will be taken, which will again try to lock the page table lock, perhaps retry page allocation because the FSDCT has now been paged in, or re-read the page if it was evicted in the interim between the time the process received the notify and the time it received the processor. The design is not to determine why the process went to wait; the hardware (by not taking a page fault) or the changed state of page control will do that on their own.

The call side (other than the process loading function) makes the decision to wait when it notices some page with I/O going on, or some PD record with an RWS going on, in a way that interferes with the contract of the entry being called. For instance, if the entry pc$cleanup is called to ensure that no pages of a segment are on the paging device in main memory (the caller having made the segment inaccessible), this surely cannot be true if there are pages being transferred into or out of main memory or the paging device; waiting for this I/O to complete is intrinsic in the contract of this entry. Similarly, the truncate function cannot destroy pages on which I/O is being performed, for the interrupt side at the completion of the I/O would have no way of telling what had happened. Leaving some kind of mark to tell it amounts to waiting for the I/O to complete.

The call side waits by calling page$pwait, with the page tables locked, passing the event ID being waited for as a parameter. Ultimately, if page$pwait so decides, this process will be made to wait. The entry page$pwait, also known as the call side wait coordinator, (its code is in the module device_control) has the following contract:

> Given a page control event ID, with the page tables locked, return when the event has occurred, with the page tables locked.

The call side wait coordinator can always decode the event ID, and by looking at a PTW or PDME, determine if the event has happened. This is the first thing it does (sees if ptw.os or pdme.rws, as befits the event, is off), and if the event has occurred, it simply returns with the page table locked, having fulfilled its contract. (It is sometimes the case that page$pwait will be called with the event ID of an event that has already happened; (see "Multiplex Wait Protocol" below.)

If the event of interest has not occurred, page$pwait decides how to wait for it in the same way as the fault side; if a PTW event for either paging device I/O or pdmap_seg, the bulk store DIM "run" entry is called in a loop until the PTW "out-of-service" bit is turned off by the bulk-store DIM's calling the interrupt side. If this is the case, the page table lock is unlocked, and page$pwait returns with it locked, having fulfilled its contract. If the event is an RWS event or a disk PTW event, the bits pdme.notify_requested or cme.notify_requested are turned on as appropriate, and control is transferred to pxss$waitp in the traffic controller. This entry unlocks the page table lock and waits for the event. When the event occurs, pxss$waitp branches to device_control$pwait_return, which relocks the page table lock (<page_fault>|[lock_ptl_no_ip]), and returns to the caller of page$pwait.

It is part of the protocol of using page$pwait that upon its return, the event might have happened, but the page is out of service again, or that it might have been fraudulently notified. All callers of page$pwait use it as part of the multiplex wait strategy outlined below; implicit in this strategy is the knowledge that these callers will retry all their operations again upon return from page$pwait. Thus, fraudulent notifications are not a difficulty. This situation is exactly parallel to that in which the restart of a page fault upon return of the traffic controller when invoked by the fault side simply retries the faulting machine cycle. No guarantee is made that it will succeed. It is the responsibility of the page control service using page$pwait to ensure that at most a finite number of retries will be necessary (see "Page State Transitions" in this section.

It is necessary that the entries used by the traffic controller to wait for page control events on behalf of the fault side and call side (other than process loading) unlock the page table lock after the traffic controller has locked its own lock. This is necessary to prevent a "lost notify" problem. Were the page table lock unlocked before the traffic controller lock were locked, the interrupt side could run in some other process, between this unlocking and this locking in real time, and the event for which the original process is going to wait will occur and be notified. Then the first process will go to the traffic controller to wait for an event that has already occurred. However, since it is necessary to have both the traffic controller and page table locks locked to perform a notify of a page control event, there is no time at which this notify might come through before the process is set waiting and the traffic control lock unlocked.

The process loading function, as stated before, causes some other process to wait than the one in which it is running. The traffic controller has a special mechanism for this, which will be explained under "Services" in Section IX. The upshot of it is as follows; traffic control will call page control to load a process. Since the process loading function cannot wait, it will either return an event ID, or, by returning zero, indicate that the process is successfully loaded. If not successfully loaded, traffic control will set the process being loaded "waiting" on the event ID returned by page control. When this process is notified, it will not be run, since it is not loaded, but rather, traffic control will call page control to load the process. Page control will either return an event ID, or the fact that the process has been successfully loaded, etc., until the process is loaded.

The process loading function calls page$pread (described in part 3 of this section) to read in the process' two critical pages. This entry calls the bulk store control "run" entry in a loop to wait out any bulk store I/O that it starts. Otherwise, this entry returns a PTW event for disk I/O that it starts, or an RWS event if one is in progress on the page. The process-loading program, wired_plm, (which is in bound_tc_wired, unlike all else in page control) sets the CME or PDME notify requested bits for each event so received from page$pread, or any PTW among those for the process critical pages that were already (or still) being read in at this call. Such a wait event is returned to the traffic controller with the assurance that a notify will be performed when that happens (this is actually using a form of the multiplex wait strategy; see that title below).

Since page control unlocks its global lock before traffic control relocks its own lock, when the process-loading function returns to the traffic controller, there is a window for a lost notify (see above). This is particularly likely on three-or-more processor configurations, where a second processor is likely to hold up the acquisition of the traffic controller lock after a third has just acquired the page table lock. There are also some lost-notify windows because the process-loading function is not in a position to apply the multiplex wait protocol properly.

It is certain, however, that if page control indicates that a process has successfully been loaded, then indeed it has. To rectify this, the traffic controller itself "validates" the nonzero event returned by wired_plm, checking the PTW out-of-service or PDME RWS bit indicated by the wait event, as required. If indeed, a notify was lost, the traffic controller puts that process in the state where yet another pass through wired_plm will be necessary to determine whether or not the process is loaded, and if not, continue the loading.

Part 2 - Multiplex Wait Protocol

As stated before, the call side of page control does not deal with individual pages at its external interface level. Calls are made to process entire segments, or deconfigure extents of main memory or bulk store, etc. All of the call-side page control entries (in PL/I page control) perform services for the rest of the system on selected groups of pages, records, or main memory page frames. Many of these functions, as noted in the above section, must initiate and/or await the completion of I/O on these various entities. The call side wait coordinator, page$pwait, is provided for this purpose.

All of these functions try to achieve a maximal degree of I/O parallelism (simultaneous I/O operations in progress). This is accomplished by processing all pages, records, or frames in the set being iterated over without performing any waiting. During this iteration, all I/Os or RWSs which need be started are started. As each page or record is processed, a check is made to see if an I/O or RWS is in progress for that page, whether or not it was just started. If this pass completes with no I/Os or RWSs found, then all of the pages or records were processed, and there is no waiting to be done, so the particular function being performed has successfully been completed. If, on the other hand, some I/O was found to be in progress, whether or not this loop had started it, the call-side wait coordinator is called with the event ID of the last such operation notified, and upon return, the entire loop retried, until successfully repeated with no I/Os or RWSs found. This technique is summarized by the following "typical" program excerpt (see any program in PL/I page control for real examples):

```
1     rt:  event = 0;
2          do  i = 0 to 255;
3              if ptw (i) meets-some-criterion then;
4              else do;
5                    call page$typical (astep, i, tmp_event)
6                    if tmp_event ^=0 then event = tmp_event;
7              end;
8          end;
9          if event ^=0 then do;
10             call page$pwait (event);
11             go to rt;
12         end;
```

The variable which is here called "event" is most often called "ind." It is often set to -1 to indicate that no code of the form of line 6 above has ever set it. The call on line 5 above performs some manipulation on a page such as starting an I/O, or continuing an eviction, etc. Such entries, all in ALM page control, perform state transitions upon pages, moving them closer and closer to the particular criterion (such as the one on line 3) which the PL/I program is trying to force to be true. Such criteria are: "No page on this PD record" (for PD record deconfiguration) or "Page not in main memory or on paging device" (for deactivation-time service) or "A good copy of the page exists on paging device or disk" (directory-unlock-time flush service, or shutdown-time main memory flush service). Such entries into ALM page control usually return the event ID of any I/O they start and do not complete, (such as page$pread, which starts page reads). A better set yet, such as page$evict and the "typical" entry above, not only return an event ID for any I/O or RWS they start, but for any they find in progress for that page at the time that they are invoked. Most do not. Some (e.g., page$pwrite) never return an event ID.

We use the term "any I/O or RWS" very loosely. Rather than being a generic class of events, any particular PL/I service (and the ALM entries it calls) might be concerned about either one or both, depending completely upon the semantics of what is intended to be accomplished.

The sort of ALM entries described above, which move pages closer to a given condition, all need some kind of prerequisite condition to ensure that no process or operation will be simultaneously trying to counteract the transitions that the ALM entry is performing. For example, the function that abs-wires portions of segments, pc_abs$wire_abs, calls page$wire_abs on each pair of segment page and main memory frame being abs-wired together, until page$wire_abs reports completion. Before ever calling page$wire_abs, however, pc_abs turns on the bit cme.abs_w, (for abs_wired) for each main memory frame in the region. The replacement algorithm will never evict a page from a frame with this bit on. No process can deactivate the segment, for only supervisor or semi-permanently activated segments are eligible for abs-wiring. Similarly, the deactivation-time service, pc$cleanup, has as part of its contract that its caller must have made the segment being processed inaccessible; thus the transitions performed by page$pwrite, called by pc$cleanup, will not be counteracted.

The PL/I loops using the "multiplex wait protocol" choose one event at random, if any have to be waited for, usually the last one encountered, and to retry the entire iteration, for at least the page associated with this event has changed states noticeably, whether or not other pages have changed state (they usually will have). Similarly, the PL/I function could not possibly be complete until that single event has happened, so it is worth waiting for it. Thus, the choice of event for which to wait is completely arbitrary. If, in fact, an earlier event were chosen, but some later call to ALM page control caused the interrupt side to be invoked and cause the occurrence of this event (post the event), the fact that this event is now invalid is of no issue, as the call side wait coordinator would discover this and return immediately, causing the loop to be redone. (No waste occurs in having the loop redone, for indeed, some I/O which was passed as "in progress" will now be finished, by hypothesis).

As stated above, the process-loading function attempts to use the multiplex wait strategy. However, instead of calling the call-side wait coordinator, which it cannot, and branching to its head, it returns an event ID to the traffic controller, expecting to be called at its entry point when that event has happened. The fact that this arrangement is not an adequate substitute for the complete service provided by the wait coordinator is obvious from the fact that events so returned must be revalidated by the traffic controller.

The various states of pages with respect to the ALM entries that cause state transitions, are illustrated in the section "Page State Transitions," along with the names of the ALM entries or the process actions that cause these transitions to occur.

DIM Interface and "Running"

Page control uses the services of two DIMs, or Device Interface Modules, to manage the I/O operations upon the bulk store and the disks. These are bulk_store_control, the bulk store DIM, and disk_control, the disk DIM.

Page control requires that these DIMs present a uniform functional interface. The semantics of this interface are one of the fundamental internal mechanisms of page control. These DIMs are known as the "Storage System DIMs," to differentiate them from printer or card punch DIMs, etc., or from the user-ring disk DIM, rdisk_.

Page control requires the storage system DIMs to have three entries, read, write, and run. The read and write entries are invoked to request the initiation of read and write operations. The run entry is used to request the DIM to interrogate its hardware status, and call the page control interrupt side if any operations have been completed.

The read and write entries are given three parameters; a device record address, a main memory address, and a word of two flags. The disk DIM read and write entries are also given a PVT index to identify the drive to which the device record address is relevant. The device address and main memory address are those to engage in the data transfer. The word of flags contains two flags, called the interrupt and priority flags. The interrupt flag tells the DIM that it is to call the page control interrupt side when the operation is completed. The priority flag may optionally be used by the DIM to sort the requests received by page control into priorities.

The Disk DIM ignores the interrupt flag, always calling the page control interrupt side. The bulk store DIM does not, however. This feature is used to write out the paging device map to the bulk store every second; as this is not really paging I/O (no PTWs or CMEs are involved), page control does not want the interrupt side to be called upon its completion.

The DIM that receives a read or write request may perform that request in any order it chooses with respect to other requests. A storage system DIM is allowed to call the page control interrupt side while processing the call to start a read or write. Specifically, it is allowed to post the completion of the very request that it was called to perform, should this actually happen. This implies that page control, on return from a call to a storage system DIM to start an operation, must be prepared to find that an arbitrary number of actions have been taken by the interrupt side during that call, including the completion of that operation.

The bulk store DIM operates entirely under the page table lock. Except when called by the Interrupt Interceptor (ii) on account of a bulk store interrupt, it is always called with the page table locked. Bulk store interrupts, however, happen only in the case of a bulk store error, in the current DIM, and the DIM itself calls to lock the page tables in each case.

The disk DIM, however, is called with the page table lock locked at some times, such as when called at the entries defined above, but not at others, as when called by the IOM manager to process a disk interrrupt. At these times, the call to the interrupt side of page control (via page$done) locks and unlocks the global lock itself.

Any storage system DIM may call the interrupt side of page control when the DIM has been invoked by an interrupt. When such an interrupt-time call is made, the DIM must itself (or via page$done) lock the page table lock, and unlock it.

The interrupt side of page control is called by the storage system DIMs with two parameters, a main memory address and a status code. The main memory address is used by the interrupt side to locate a core map entry, from which all other information (such as cme.rws, for example) may be derived. The status code indicates the degree of success of the I/O operation. The low bits of the status code indicate to page control the DIMs determination of whether the problem causing the error is an error in the device, the data path to the device, the record of the device, or the page frame of main memory involved in the attempted transfer. Page control uses this for error recovery (see "Error Strategy" below).

A DIM may retry an operation it has been asked to perform any number of times; page control is only interested in the final outcome. What is more, a storage system DIM can write some page to any number of different records or devices, as it sees fit, and when asked to read it back, read it from any (or all) of them. It is guaranteed by page control that all such copies will be "good." If page control detects that the page was modified when an attempt at eviction is next made, none of them are good; if not, they all are. What is more, a storage system DIM can use the main memory frame into which it is being asked to read for any intermediate buffering, diagnostic results, etc., as long as it contains what was asked for when the operation is posted. Page control makes no assumptions about the contents of page frames that are out of service on reads.

If a storage system DIM given a request to perform, finds that it has no queue space, it is allowed to loop internally awaiting the real-time completion of I/O requests on its devices, possibly calling the interrupt side of page control, if that is necessary to free queue space.

A DIM is allowed to perform services for other parts of the system, as the disk DIM does for the VTOC manager, possibly calling the page control interrupt side when so doing. In such cases, this call must be treated like one on behalf of an actual interrupt.

A DIM must provide a "run" entry, called only by page control with the page-table lock locked, which checks the devices being managed for operations that have completed, and calls the interrupt side of page control for any that have. Such an entry must have two properties:

1. It must physically interrogate the hardware status (perhaps stored) of its device; it cannot depend upon actual interrupts having happened to take cognizance of I/O completions.

2. If called in a loop, I/O operations will be posted one by one via calling the interrupt side of page control, until the DIMs queues hold no more uncompleted requests.

For one example of the use of a "run" entry, see the previous section, where the page fault handler calls the run entry of the bulk store DIM until it finds that the I/O on the faulting page has completed.

The RWS initiator (rws_ in pd_util) "runs" all of the DIMs (calls their "run" entries, one by one, for all two of them) in a loop when more than thirty RWSs are awaiting completion. Thus, it is guaranteed that doing this arbitrarily long will cause an arbitrary number to complete.

The paging device replacement algorithm runs the bulk store DIM to make sure that all read cycles are finished before it is exited, and the page table lock potentially unlocked.

The main memory replacement algorithm runs the DIMs in a loop if an excess (currently 30) of uncompleted page-write I/O requests are outstanding. (The tool file_system_meters reports occurrences of this event.)

The traffic controller "polls page control" every 15 seconds, which consists of calling page$run, which locks the page table lock, runs all the DIMS, and unlocks the lock. This, as all run calls and all other calls, may be used by the DIMS to perform timing-out functions and housekeeping.

Other than calling the bulk store run entry as a substitute for traffic control waits, no page control module other than the ALM program device_control ever calls the storage system DIMs directly. Rather, the entries device_control$dev_read, device_control$run and device_control$dev_write are called. These entries, called only from the ALM page control environment (PL/I page control never deals at this low a level), use variables in the ALM page control environment to determine which DIM to call. This is the function, and the origin of the name, of device control. The call-side wait coordinator also resides here, as well as the page control code called as "page$run" which runs the DIMs on behalf of the traffic controller polling code.

ALM Page Control Environment

All of the ALM programs in page control, including the bulk store DIM, share a common environment of register usage, and all share the same stack frame while in the same invocation of page control. That stack frame is laid out in pxss_page_stack.incl. alm. As can be inferred from the name, the traffic controller shares the same stack layout, which is meant to optimize the case where page control calls or transfers to the traffic controller; in this case, the actual stack frame is shared.

Almost all subroutines in the ALM page control environment invoke each other via the TSX7 instruction; there are a set of "small" subroutines that are invoked with a TSX6 instruction. A subroutine is "small," if it calls no other subroutines.

Any subroutine that calls any subroutine except a "small" TSX6 subroutine must do a "savex"; this operation, performed by the "small" TSX6 subroutine of that name saves index register seven in a stack of saved values in the stack frame. This stack is initialized by the routine init_savex. A subroutine that has not done a "savex" returns via TRA 0,7. One that has returns by branching to the code "unsavex," which pops the stack and returns.

All code in ALM page control, other than the bulk store DIM, runs with pointer register 3 set to point to the base of the SST. Any code that exits the ALM page control environment must restore it.

All external entries to the ALM environment, such as the page fault handler, and the entries called by PL/I code (through the transfer vector "page") are responsible for setting up this environment, i.e., initializing the index register save stack and pointer register 3.

Other than these general conventions, there are conventions of dealing with specific data objects. When any ASTE, PDME, PTW, or CME is being dealt with in any way, all routines expect the following index and pointer register assignments to hold:

Object	Register	Symbolic Name
PTW	Index 2 and Pointer Reg 2	.ptw ptw
CME	Index 4	.cme
ASTE	Index 3	.aste
PDME	Index 1	.pdme

The values in the index registers, are all offsets relative to the base of the SST (pointed to by pointer register 3, symbolically "sst"). These symbolic names are used by most code in the ALM environment to reference these registers. Pointer register 3 also has the names "cme" and "ast" and "pdm" to allow references of the following form to be made:

```
lda     ast|aste.uid,.aste
```

These symbolic register names may be found in the include files page_info.incl.alm and page_regs.incl.alm.

The use of the stack variables in the ALM page control stack frame is not systematized in any way. No person attempting to modify or maintain page control should change any routine to use any variable that it had not previously used unless they are familiar with every single use of that variable in ALM page control. No attempt is made to document the usage conventions of these variables. This can only be learned via extensive experience with ALM page control. The only variables of any general interest are those named "devadd," "coreadd," "did," "errcode," and "inter." The variables "devadd," "coreadd," "did," and "inter," are the record address, PVT index, and Flag word, respectively, passed to the storage system DIMs. Bulk store control, sharing the same stack frame, uses them in place. The variables "coreadd" and "errcode" are used by the interrupt side to receive the descriptions of completed operations. Again, the bulk store DIM uses them in place. It is also worth mentioning the array "arg," which is used by both page control and traffic control to prepare argument lists and descriptors for any external (PL/I) call that must be made from the ALM page control environment.

Error Strategy

By "error," we refer to any of the following three types of abnormal circumstances:

1. Those resulting from user behavior (e.g., record quota overflow).
2. Those resulting from I/O device error.
3. Those resulting from internal software, or processor error.

The first class of error situation can hardly be considered an error situation at all. The only "errors" in this class are physical volume overflow and record quota overflow. Both of these errors are detected on the fault side; supervisor segments are quota-inhibited (aste.nqsw is on) and prewithdrawn, making these classes of problems impossible. Should they occur on a supervisor or semi-permanently active segment, the system software is malfunctioning, and a class 3 error results. Record quota is checked by the fault handler before any quota cells are incremented; availability of physical records is similarly checked by the record allocation function (free_store) before any data bases are modified. Thus, recovery from either of these circumstances involves no "backup." Record quota overflow is signalled by the fault side on the stack on which the faulting process was running. This is done by moving the page fault machine conditions to pds$signal_data, abandoning the masked/wired environment, and transferring control to "signaller," the procedure responsible for effecting such signalling.

This causes the stack history on that stack to be such that a return to the signaller's frame causes the page fault to be retried. (This is the standard fault-signalling, the only difference here from the common case being that a masked, wired environment, with a stack frame on the PRDS (wired stack segment) was abandoned.) Physical volume overflow is handled by the fault side by marking the ASTE of the segment (aste.pack_ovfl) for which a record cannot be allocated, setting a segment fault in the SDW for the segment implicated by the page fault machine conditions, and restarting the fault. This causes a segment fault to be taken. The segment fault handler locates the ASTE, sees the bit, and invokes the segment mover, presumedly resolving the physical volume overflow situation (see "Segment Moving" in Sections II and IV).

The class of errors produced by detected I/O device failure is that one in which page control policy has the greatest effect upon system behavior. Errors are reported by the storage system DIMs (see "DIM Interface," earlier) to the interrupt side of page control. This severely limits the actions that can be taken at that time. Specifically, no operation that involves waiting can be performed. Furthermore, since the interrupt side can be activated by the call side whenever a DIM is invoked, no action that involves allocating main memory or paging device frames is permissible, since that would involve all of this software recursively. This class of errors may be further subdivided into errors in reading and errors in writing.

Errors in reading are simpler to handle, because there is always some process waiting for the completion of that read. Taking whatever action is necessary and notifying an appropriate event will cause that process to retry that read, either via the fault side retry mechanism or the call side multiplex wait protocol. The response to disk read errors is to turn on the bit ptw.er in the relevant PTW, and return the PTW (otherwise) to its original state before the read was started. Subsequent notification of the associated event causes the fault side to retry, notice the bit, signal an error (condition page_fault_error) (via the same fault-side signalling mechanism as is used for record quota overflow), turning off this bit while so doing. The next retry of that page fault causes another attempt to be made at the disk read.

Errors in reading the paging device (on other than RWS read cycles) are much the same. However, the paging device record involved is dynamically deleted by the interrupt side, because of the fact that an error was encountered in reading it. A syserr message accompanies this action. The disk address (possibly nulled) which was in the PDMAP entry (pdme.devadd) replaces the PD record address (nptw.devadd) in the PTW, causing the next retry of this page fault after the one that signals error to obtain the copy of the page on disk (or zeros if the address in the PDME was nulled).

Errors in reading the paging device for the read cycle of an RWS are somewhat like paging device errors above, although a different error message is printed by syserr. The paging device record is dynamically deleted, and the (possibly nulled) disk address in the PDME replaces that in the PTW. Since, by implication, the RWS has been declared over on account of that error, and the data on disk is thus considered implicitly "valid," the main memory frame of the RWS is freed, and there is no write cycle. No resurrection of disk addresses is performed in this case. Errors during RWS on behalf of the post-crash PD flush are discussed in the consideration of that mechanism in Section IX.

Errors on writing are difficult to handle. While the optimal policy would be to allocate a new disk or PD record, this requires manipulation of the paged segment FSDCT, which is impossible at interrupt time. For the case of write errors to the paging device, the solution is simple; the relevant paging device record is deleted, and the (possibly nulled) disk address from the PDME replaces the PD address in the associated core map entry (CME). This has the effect of forcing the replacement algorithm, or the call side, on behalf of whatever agency is trying to see the completion of this writing, to retry writing, accomplishing a write to disk instead. In effect, the page has been migrated off the paging device.

Errors on writing to disk are problematic in the ways stated. The action at this time is to replace the disk address associated with the page with a null address (page_bad_null, see null_addresses.incl.alm), freeing the main memory frame, causing the contents of the page to become zeros. Errors on writing disk on behalf of the write cycle of an RWS are similar; the null address page_bad_pd_null replaces the PD address in the PTW, and hence, ultimately in the file map. No resurrection, clearly, is performed. Again, special action is taken for the post-crash PD flush.

The third class of errors dealt with in page control is that class of errors indicating software malfunction. In every case, it is dangerous or impossible to continue system operation, since further damage and wrongly disclosed data would probably result. Included among such errors are errors found in locking, errors in expected states of data bases, errors in threading, and so-called "re-used addresses" (records marked as free that are known to be in use, or being freed). Such errors can result only from undetected processor or main memory malfunction, or undiscovered bugs. The effect is to crash the system in every case. In PL/I page control, this is accomplished by calling syserr explicitly. In the ALM environment, the routine page_error is responsible for constructing and executing all syserr calls. There are some entries to this routine (including those used by the bulk store DIM) that report specific errors (such as the non-fatal read and write errors, and paging device record deletions discussed earlier). These routines are knowledgeable about stack variables in the ALM environment, and variable information is printed out in their messages. There are also some entries that crash the system with a specific message, such as that which is invoked upon discovery of a reused address. However, the most commonly used entry is that invoked from the routine page_fault_error in the program page_fault. This routine is invoked from the ALM page control environment via a TSX5 to page_fault_error. It crashes the system with the error "fatal page fault error at location xxxx" where xxxx is the address (in bound_page_control) of the TSX5 instruction executed. In every case, this type of crash is the result of software malfunction, possibly induced by undetected hardware failure. (There is also one case of this type of crash induced by detected hardware (processor) failure; that in which no appending unit cycle bits are on in the page fault machine conditions, indicating appending unit failure.)

There is also a "nonfatal page fault error" facility, which is very sparsely used.

Calls to crash the system via the program page_error call the PL/I routine syserr via a standard call, setting up their argument lists in the array "arg" in the page control stack frame. Part of that PL/I call is the storing of all of the index registers and the AQ at location 40 (octal) in the stack frame of the ALM environment; useful information about the data objects invoked in such a crash can always be gleaned from this data.

The crash for a re-used address is peculiar insofar as the code that invokes it turns on the bit pvte.vol_trouble before crashing. This action causes the physical volume whose volume map was involved to be volume-salvaged the next time it is accepted for storage system use.

Other than these errors, there are no possible errors in page control. No call side entries, or entries to ALM page control return a status code of any kind. No nonfatal failure is possible in the current design. However, in some cases, such as RWS failure due to I/O error on behalf of the post-crash PD flush, status information is conveyed back via the live/nulled/null status of the address left in the PDME by the RWS interrupt side. (See the description of this service in Section IX.)

Stack Management and Interface with the Traffic Controller

Page control uses the wait/notify facility of the Multics traffic controller fairly heavily. The conventions for such waiting and notifying have been discussed.

Page control does not notify any event unless some process is waiting for it, in order to avoid the overhead of traffic control. The bits pdme.notify_requested, cme.notify_requested, and pdme.abort fulfill the function of specifying whether or not such notification is to be performed. All notification is done by the interrupt side, in ALM page control (save for one highly esoteric case during boundsfault processing; see Section IX). All waiting is also performed by ALM page control; the primitive page$pwait serves to perform such waiting on behalf of PL/I page control. The mechanism used by process loading to wait has already been discussed.

The interface between page control and traffic control is streamlined to facilitate these operations. Since the traffic controller and ALM page control share the same stack frame layout, with variables in it allocated to each, the interrupt side transfers directly to a special side-door entry to the traffic controller (pxss$page_notify or pxss$rws_notify) to perform all such notifications. The traffic controller returns to the side-door entries to the procedure page_fault (page_fault$notify_return and page_fault$rws_notify_return) after notifying. The event ID to be notified is passed by page control in the cell pds$arg_1. The quantity seen in the listings as being passed in pds$arg_2 is an obsolete remnant of an old device-metering mechanism. The traffic controller operates completely in page control's stack frame in these cases.

The wait interface is more involved. The interface used by the process-loading function is not discussed here; this has already been treated. The traffic control interface for waiting is always invoked by ALM page control via a direct TRA, from either code in the end of the page fault handler, for (invoking pxss$page wait) causing a process to wait on behalf of the fault side, or from page$pwait, the call-side wait coordinator (invoking pxss$waitp). There is also a third entry, pxss$ptl_wait, used explicitly by the fault-side mechanism that allows multiprogramming to wait for the page table lock. Other than this third mechanism, these entries are entered with the page table locked in every case, being unlocked by the traffic controller after its own lock has been unlocked (see "Wait Protocols" earlier, for the reason this is done).

The interface invoked by the fault side, pxss$page_wait, shares a stack frame from the PRDS with the fault side, which invoked it. The fault-side stack frame <u>becomes</u> a traffic controller stack frame, on the PRDS, and is managed by the traffic controller from that point on as a traffic controller PRDS stack frame, as it is passed around from process to process. Entry to the traffic controller via pxss$page_wait implies that the entire state of the invoking process is encoded in the page fault machine conditions in pds$page_fault_data in that process; this is to say that there is <u>no</u> page control stack history of any kind in that process. Thus, when a process waiting via this mechanism is notified, and subsequently allowed to run, the traffic controller <u>transfers</u> to page_fault$wait_return, which does nothing more than restart those machine conditions (including process/processor mask). Specifically, the page table lock is not locked, nor are any page control data bases at all inspected or modified in any way. This causes the faulting machine cycle to be restarted, either completing successfully (if the page fault has been resolved) or taking another page fault.

When the traffic controller is invoked to wait on behalf of the call-side wait coordinator, a transfer to the entry pxss$waitp is effected. Again, pds$arg_1 contains the event on which it is desired to wait, and the page table lock is locked, to be unlocked by the traffic controller. When a process waits via this mechanism, PL/I page control has a stack history on the PDS of the waiting process; the stack frame that was the current stack frame of that process contains the return pointer to the place in the PL/I program that called page$pwait; that point must be returned to when the waiting has been finished. There are no machine conditions; action upon return from the traffic controller consists of transferring to that place in the PL/I program. Thus, the traffic controller, upon completion of such waiting, transfers to the side-door into the wait coordinator, device_control$pwait_return. Since the page table lock has been unlocked, this entry relocks it via a call to the ALM page control locking interface (page_fault$lock_ptl_no_lp), and returns to the PL/I program at the instruction after the call to the wait coordinator. In order for this policy to succeed, the stack frame pointer register (Pointer Register 6) must be restarted at the time device_control$pwait gains control, to its value at the time that pxss$waitp gained control. Therefore, the traffic controller saves this value in the cell pds$last_sp, which is often useful in debugging problems in this area.

The traffic controller differentiates between the two cases above (fault side wait, no stack history, and call side wait, PL/I PDS stack history) via the variable pds$pc_call, zero for the first case and a positive nonzero number for the second. The value of this variable tells it whether the state of a process waiting for a page control event is embedded in the machine conditions in pds$page_fault_data, or in its PDS stack history, as defined by the value of pds$last_sp. This implicitly tells it whether it should transfer to page_fault$wait_return or device_control$pwait_return.

The mechanism used to wait for the page table lock on the fault side uses exactly the same mechanism as used by the fault side to wait for other events. A special entry to the traffic controller is used in this case (pxss$ptl_wait), which performs certain manipulations as described under "Page Table Lock Waiting" later in this section. However, this special code soon transfers to the code used by the fault-side to wait for all other events. Thus, it is to be noted that the action performed upon notification of the page table lock event is simply to retry the page fault, just like any other fault-side wait.

The variables, pds$last_sp and pds$pc_call, are used by the traffic controller for other mechanisms than page control waiting. Specifically, pds$last_sp is used for all calls to the traffic controller for waiting (other than those just described). The cell pds$pc_call is also used by the traffic controller's preinitialization and shutdown wait mechanism (pi_wait) to differentiate other wait calls than page control's from the two kinds of page control wait already discussed; in this case, pds$pc_call is set to a negative value.

See Figure 8-1 for a synopsis of this mechanism.

In all cases of invocation by ALM page control, the traffic controller is aware that the process/processor are masked to "sys_level," and all relevant parameters are in wired storage. Thus, the traffic controller never pushes its "extra" PDS frame in these cases, because it is used only to store old masks.

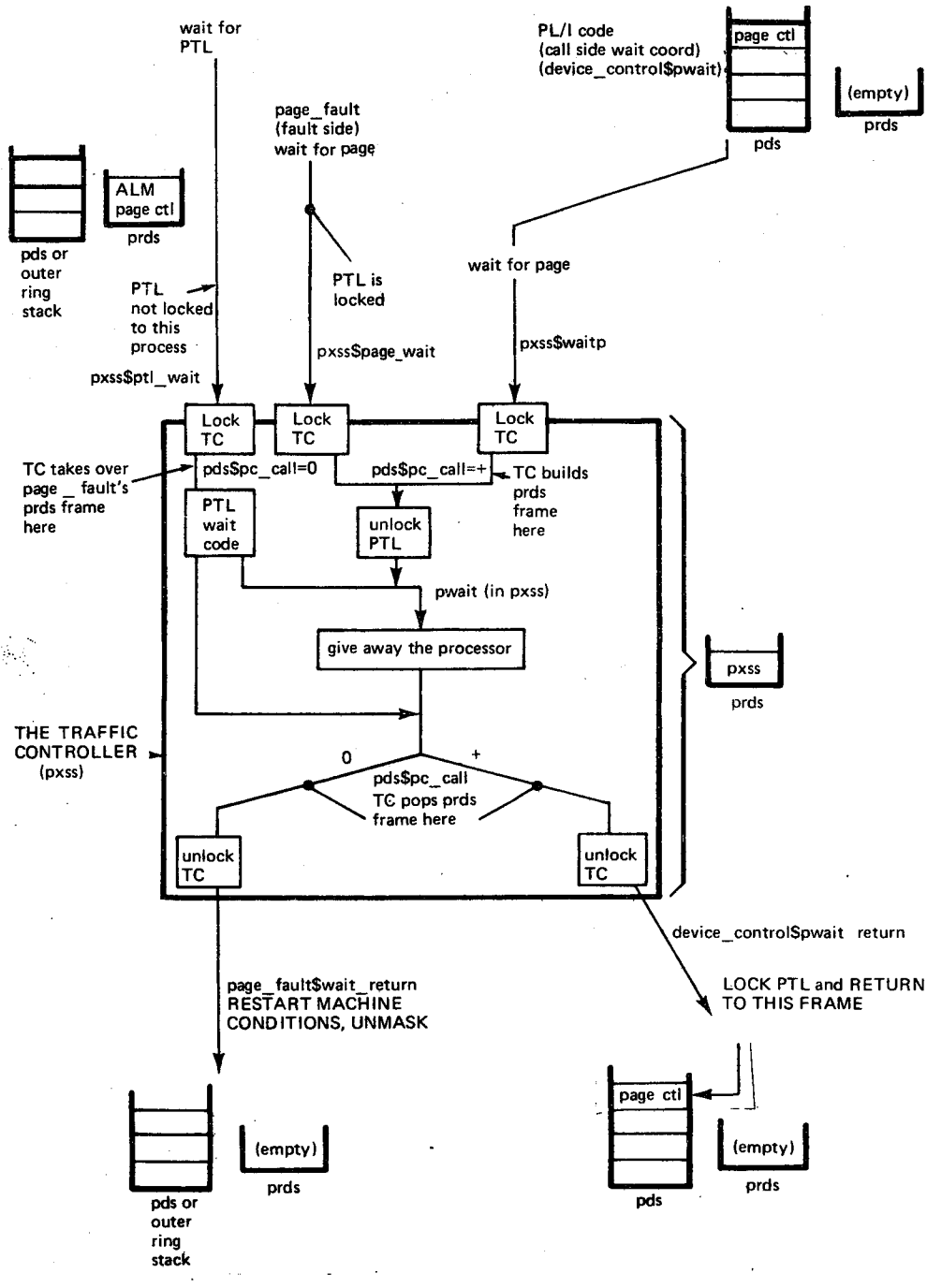


Figure 8-1. Traffic Controller Interface Stack Management

The external entry to page control to lock the page table lock does not need a stack frame; it does not push one (page$lock_ptl, using lock_ptl_no_lp in page_fault). The external entry to unlock the page table lock, however, does, because the traffic controller may be invoked to notify the page table lock event. It pushes its frame, and does a full return (page$unlock_ptl, invoking unlock_ptl, in page_fault). A special side-door is used by privileged_mode_ut$unlock_ptl, however, to avoid pushing a frame. This side-entry page_fault$pmut_lock_ptl, pushes a frame, and explicitly pops it in line before transferring privileged_mode_ut$unwire_unmask to finish the job.

Note that all side doors to page control go directly to individual ALM programs, and not through the transfer-vector "page."

Page States

One instructive perception of page control is that of a set of finite-state automata; one for each page, one for each main memory frame, one for each paging device record, and one for each secondary storage record. The basic operations of page control, specifically the actions performed by ALM page control, consist of performing state transitions upon these objects. PL/I page control, via iteration and the multiplex wait protocol, effects many state transitions at once.

A series of diagrams (Figures 8-1 to 8-7) presenting the various states of these automata is presented here. The entry points, code sequences, or actions that affect each transition are identified. The flags and fields that define each state are identified.

In almost all cases, state transition is performed under the global page table lock. Almost all states of these pages and records are valid when the page table lock is not locked. A notable exception is the read cycle of a paging device read-write sequence (RWS) that is only seen under the protection of the page table lock.

Refer to the discussions of the page table lock strategy and the multiplex wait protocol for more illumination on the motivations for these sequences.

Special mention must be made of the illustrations, Figures 8-2 and 8-3 which show the state transitions of the page of a segment. To avoid over-complication of the diagram, transitions involving paging device deconfiguration (manual and automatic) have been omitted. Also omitted are the data base bit states that denote these page-states as well as program names; again, to avoid over-complication of the diagram. The state of all bits may be inferred from the three previous diagrams. All of the transitions marked "modification" in these diagrams represent not the action of any sequence of code, but rather, that of the user of a given page, in modifying the contents of that page. The transitions and states relating to the use of a page, only of interest to the main memory replacement algorithm, have not been shown in Figures 8-2 and 8-3.

Figure 8-2 is divided into two regions, states of a page that have no associated paging device record, and those which do. The later region is further divided into two regions, those in which the paging device page copy is identical to the disk copy ("PD Notmod") and those in which it differs ("PD Mod").

Two recurring patterns can be seen in each of these regions, the "read-evict" cycle, in which a page is paged in from main memory, used without modification, and evicted; and the "write-mod" cycle, in which a page in main memory is modified by use, written out to "purify" it, and brought thus back to the in-main-memory state of the "read-evict" cycle. When such cycles are isolated, Figure 8-3 is the result, showing the states of a page with respect to main memory and the paging device in terms of these cycles. An entire set of such cycles is shown in Figure 8-4, including the "used" states of interest to the main memory page replacement algorithm.

It should be noted that there are no states in any of these diagrams corresponding to the sematics of the bits cme.abs_wired, ptw.wired, pdme.removing, cme.removing, and pdme.flushing. These bits do not represent states per se, but rather instructions to all of the routines that perform the various state transitions as to a desired "goal state." For instance, the flag ptw.wired inhibits eviction, i.e., transition out of those page states where the page is in main memory. The flag cme.abs_wired not only prevents eviction via the replacement algorithm, but any subsequent assignment of the main memory frame to any use (transitions out of the "free" state in Figure 8-5) by any code except that of the abs-wiring function. Thus, the PTW "wired" bit is turned on at any time (see earlier discussions of the page table lock), with the knowledge that any subsequent read-in of the page will cause it move to the "in-main-memory" state and stay there. For the case of wiring a page, the transition to the in-main-memory state is easy to force simply by touching (i.e., faulting upon) the page. In other cases, such as demand eviction on behalf of memory deconfiguration, this is substantially more complicated. Thus, primitives such as evict_page (in ALM page control) exist which, given the appropriate data objects (in this case, a core map entry representing a main memory frame), with such bits already turned on, perform whatever transitions are necessary to achieve the desired state (in this case "free"). If the transition is to or from a state where I/O is performed, a wait event ID is returned, otherwise the complete transition is made, and no event ID is returned. The greater part of call-side services such as the abs-wiring and main memory deconfiguration services is to turn on such bits, and call such primitives on each subject page and/or main memory frame repeatedly, multiplexing indicated waits via the multiplex wait protocol.

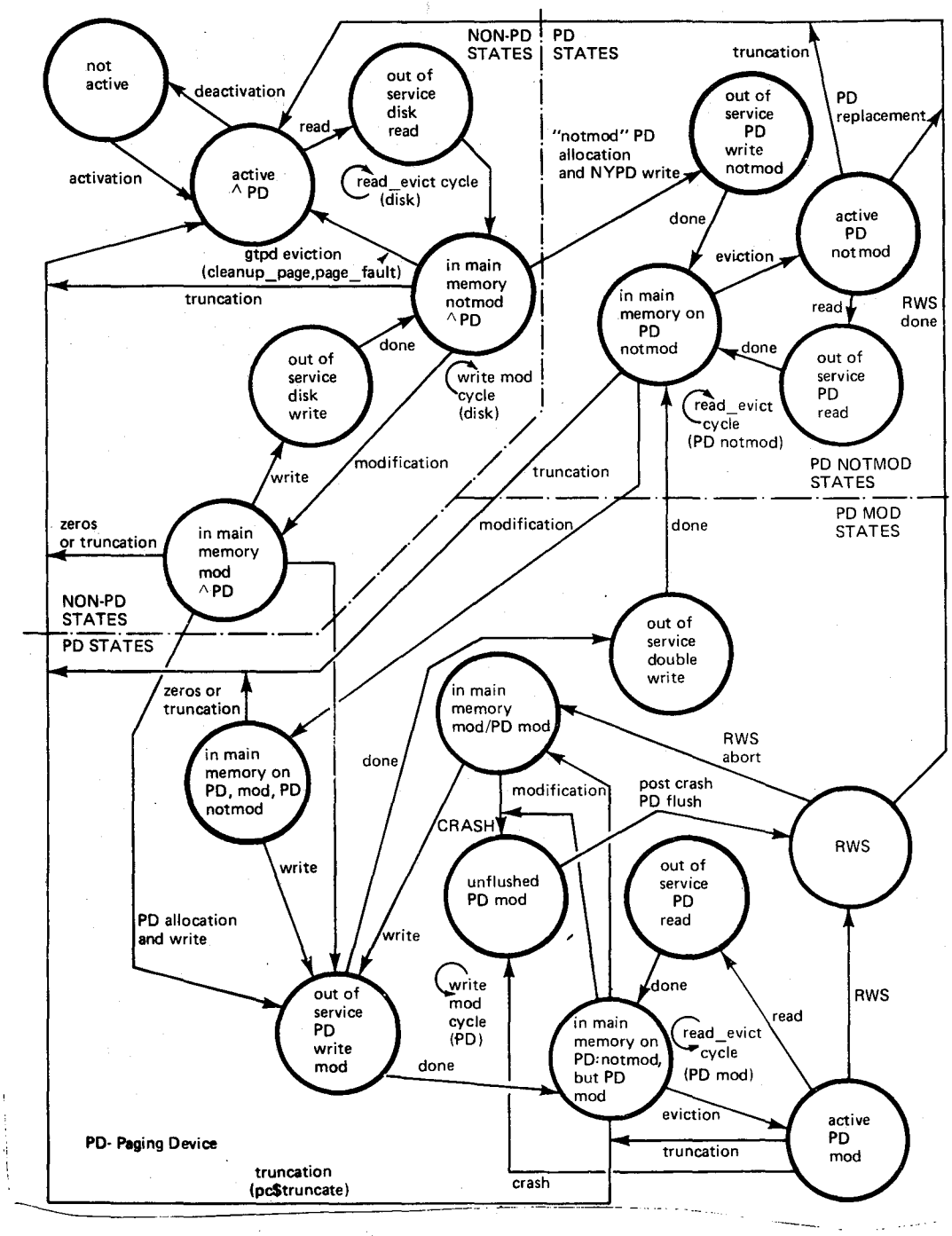


Figure 8-2. States of Page

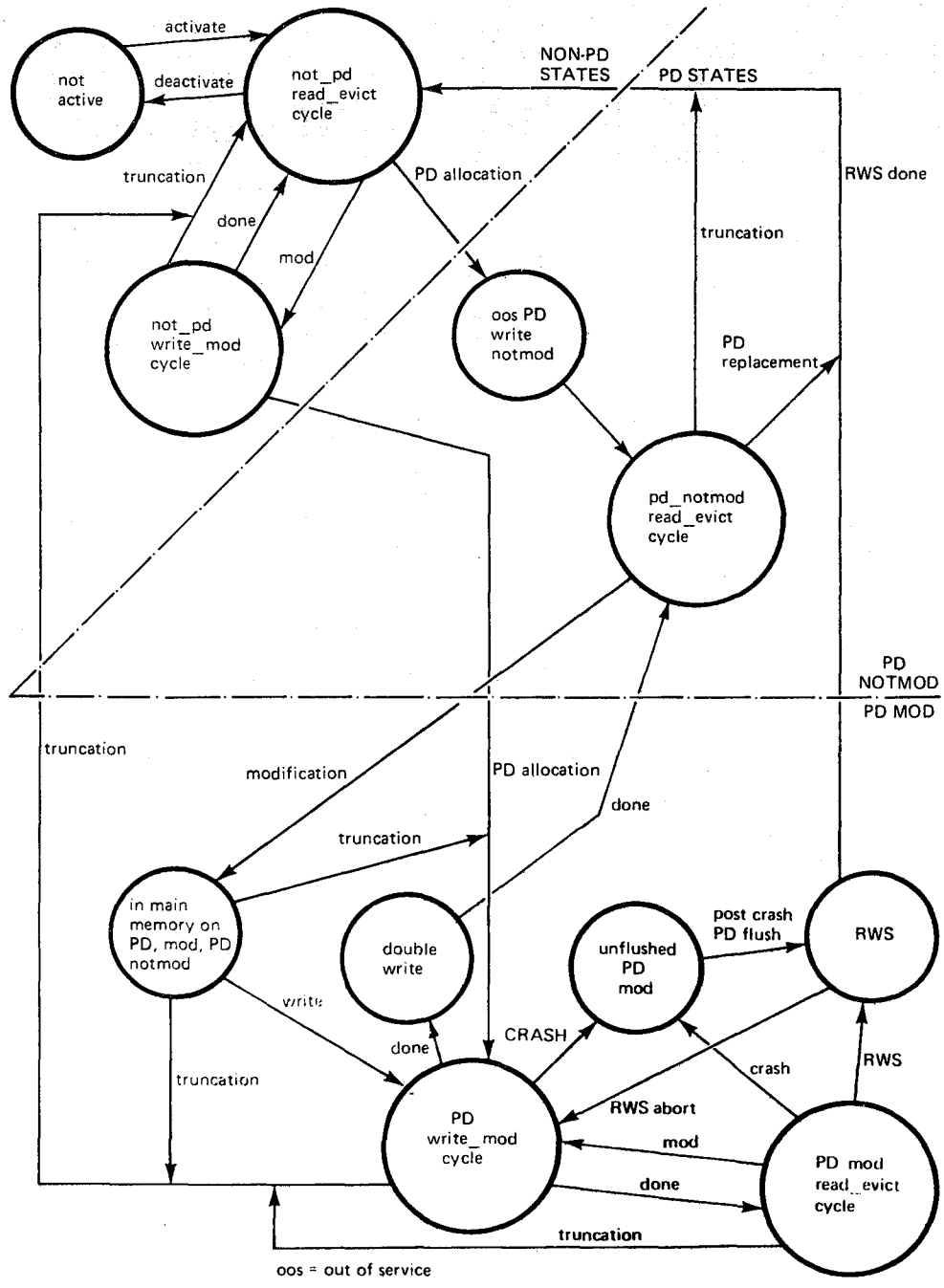
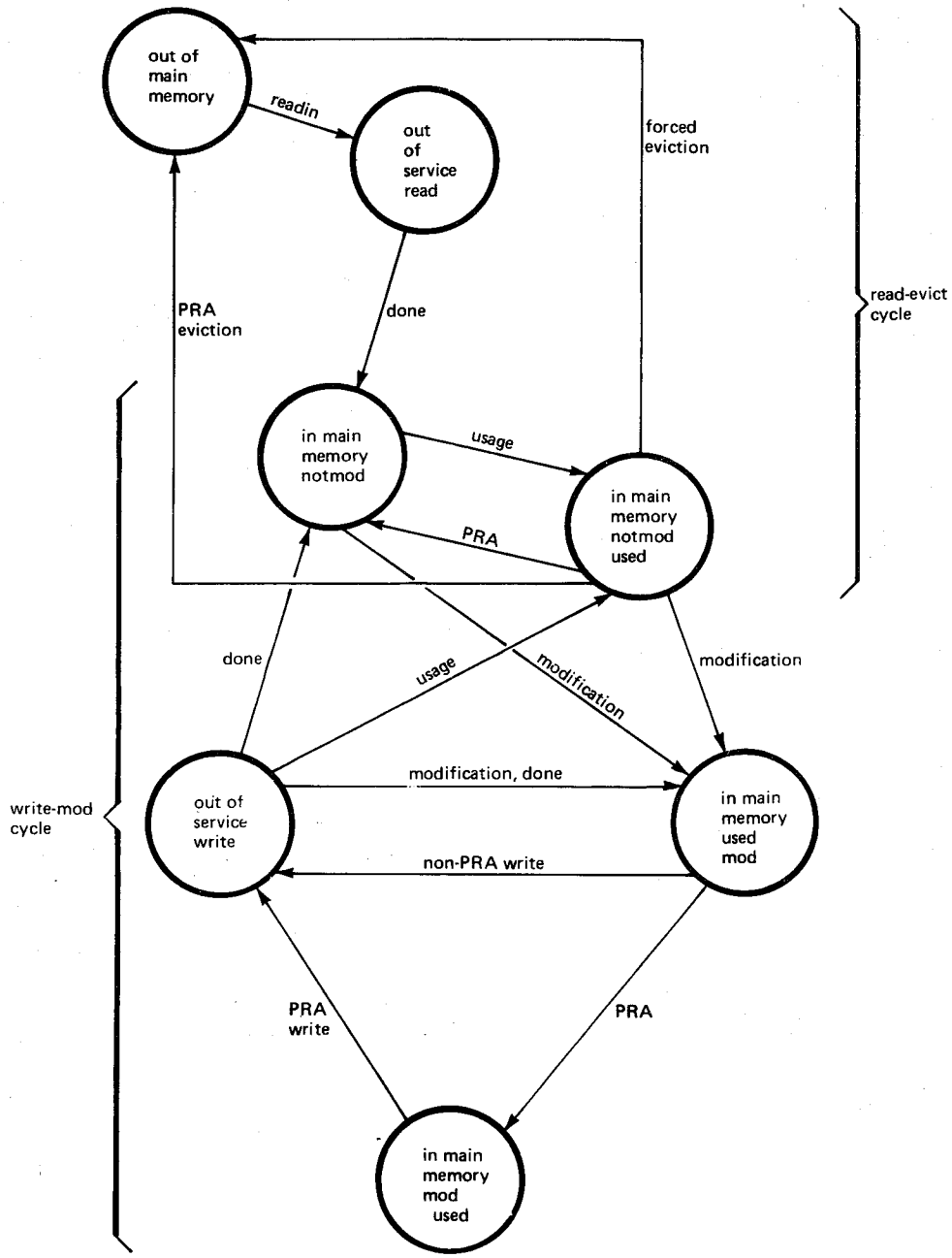


Figure 8-3. States of Page in Macro States



PRA=main memory page replacement algorithm

Figure 8-4. Read-Evict, Write-Mod Cycles

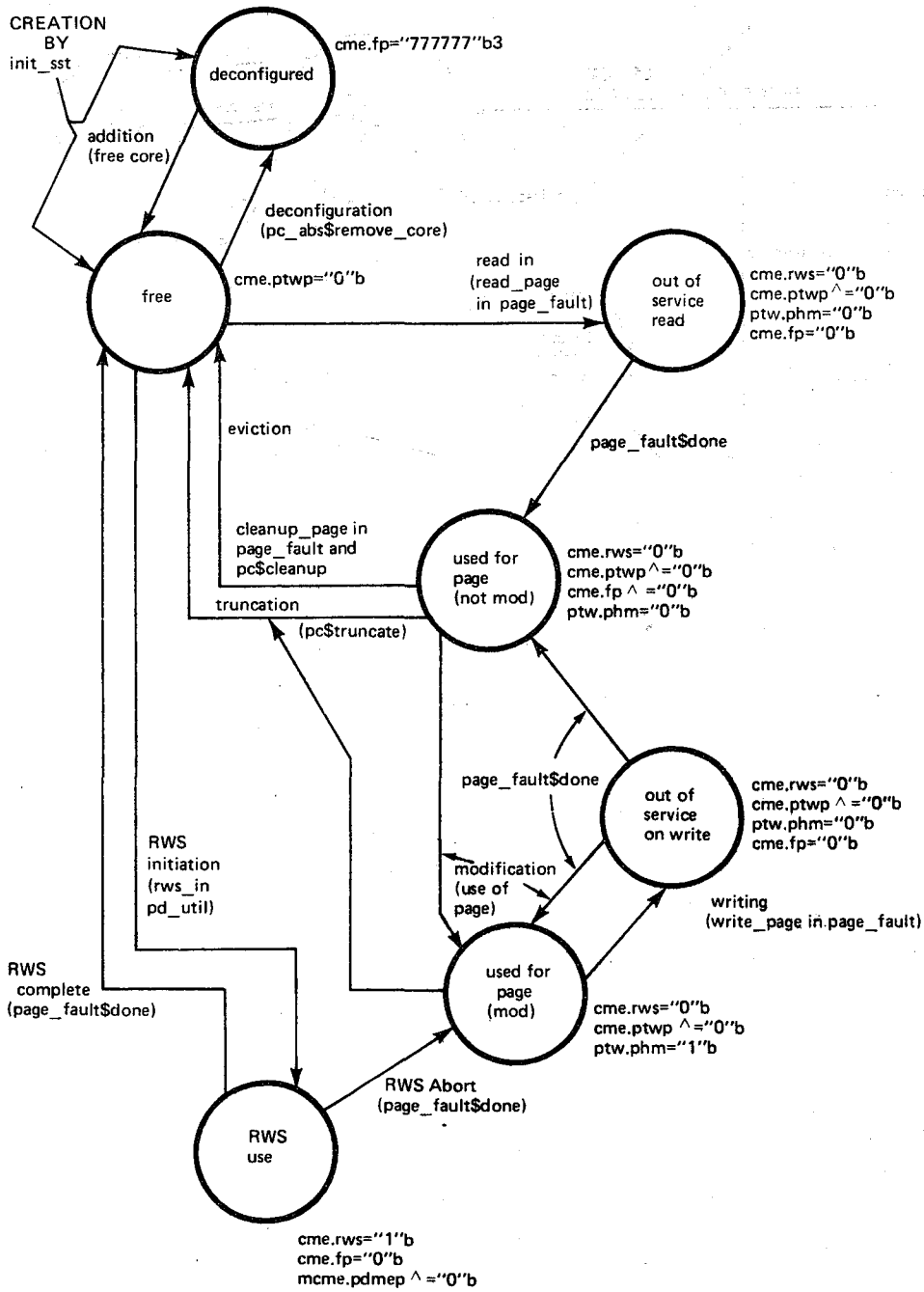


Figure 8-5. States of Main Memory Frames

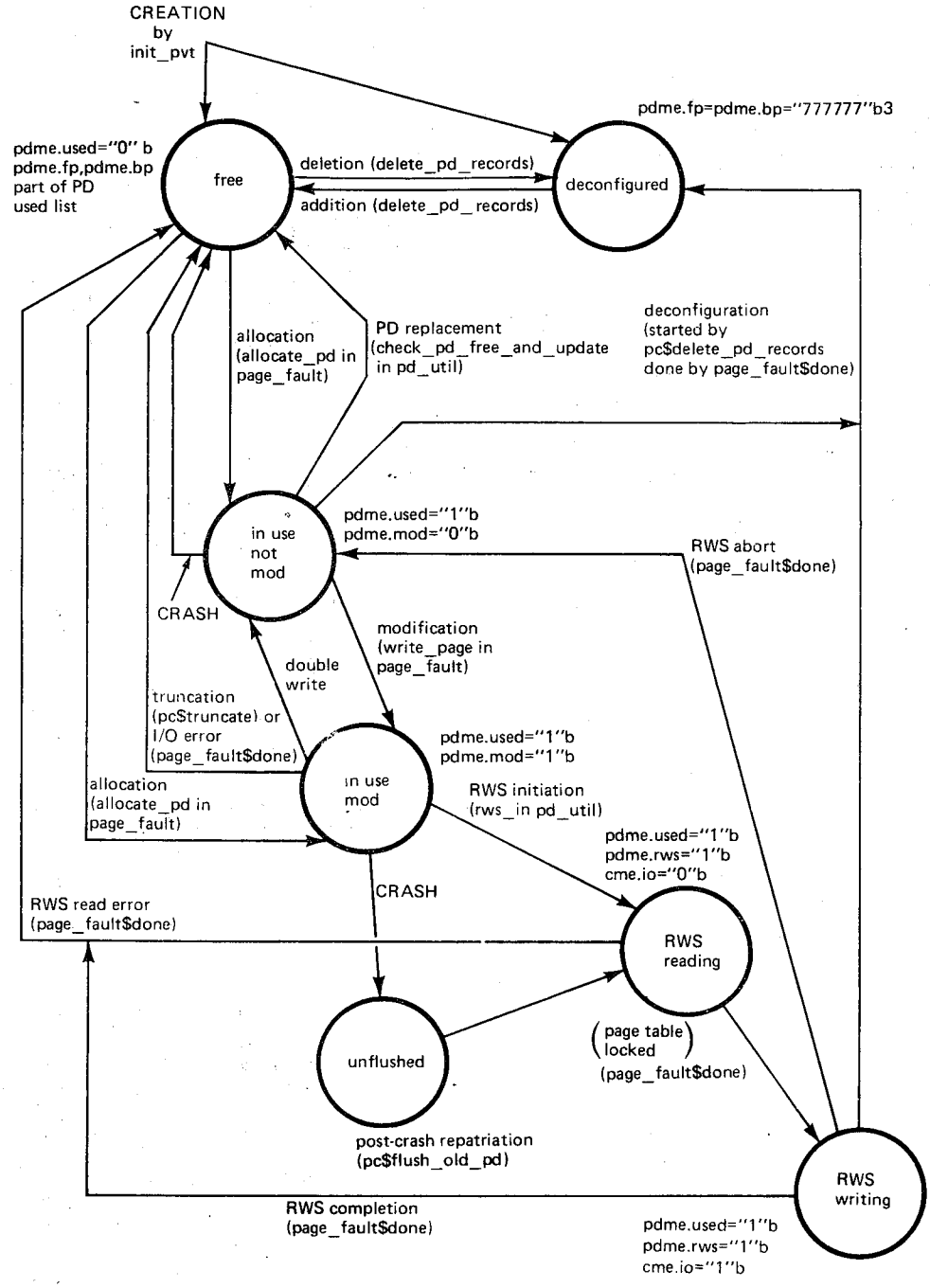


Figure 8-6. States of Paging Device Record

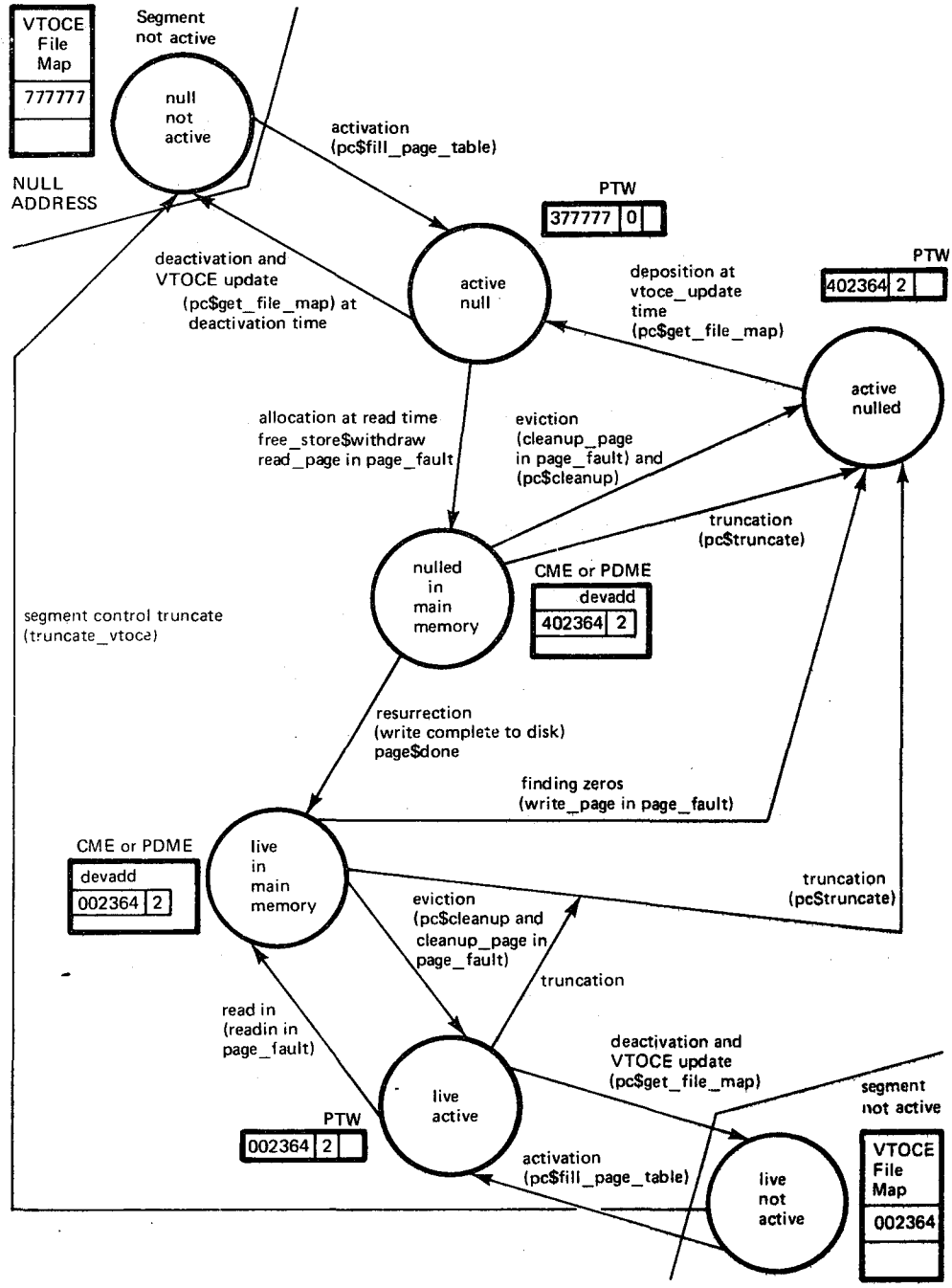


Figure 8-7. States of Disk Address

Tracing Mechanisms

There are two tracing mechanisms in page control, both of which have not been maintained in recent years.

The "page control trace" mechanism is part of the hardcore system trace facility. It is enabled by switch 34 on the Processor Maintenance Panel switch register. This switch register is read and stored in sst.trace_sw on every page fault. When enabled, this trace facility causes tracing messages to be printed or written to tape, as selected by the system trace facility Various callside routines (mainly to the program "pc") inspect this switch, and call "trace," the system trace routine, with arguments describing the action being performed, and the location and contents of the AST entry upon which they are being called to operate. Many actions in ALM page control are traced as well; they can be located via the calls to "pc_trace" in ALM page control.

The program pc_trace is part of ALM page control. It is invoked at its various entries, each of which traces one type of event, via a TSX7 instruction from within bound_page_control. This program issues no messages; rather, it sets up argument lists for the program pc_trace_pl1 which does. These argument lists are functions of the individual entries. The actual arguments are particular stack variables and index register values from the invoking ALM page control environment. The program pc_trace_pl1 contains nothing but the PL/I calls to the system trace facility, referencing the arguments passed by pc_trace.

The second trace facility in page control is that referred to internally as "disk_meters." This facility is the remains of an experiment described in the MVT Project MAC Technical Report cited in Section V, which accumulated traces of paging device allocations and evictions in order to achieve performance predictions for extended paging devices and main memories. This facility is enabled and disabled via the program "get_disk_meters," which wires and unwires the tracking buffer, "disk_traffic_data." The trace entries are accumulated by the program "meter_disk," invoked from ALM page control via a TSX0 instruction. All entries to this procedure start with an XEC instruction; when this facility is not enabled, the target of this instruction is a TRA 0,0, which returns at minimal cost. This facility has not been functional since release 4.0; furthermore, there are no installed tools to retrieve or interpret its output.

INDIVIDUAL MECHANISMS

Waiting for the Page Table Lock

The fault side of page control has the ability to utilize the traffic-controller wait/notify mechanism to wait for the page table lock to be unlocked. This ability depends upon the fact that the fault side has not modified any data bases or changed the state of its process at the time that it encounters the page table lock locked. Thus, if that process is made to wait for the unlocking of the lock, via the traffic controller, the return from that wait may simply restart the machine conditions of the page fault, probably taking the page fault over again and retrying the operation. Thus, it may be seen in Figure 8-1 "Traffic Controller Interface Stack Management" that the entry to the traffic controller to await a page from the fault side (pxss$page_wait) ultimately merges with that which awaits the page table lock's unlocking (pxss$ptl_wait), both returning to page_fault$wait_return to restart the fault.

Since processes may be waiting for the unlocking of the page table lock, it is potentially necessary to notify the "PTL event" (the page table lock event ID, "160164153152"b3) every time the page table lock is unlocked. Since there is a substantial overhead involved in calling the traffic controller notify primitive to do this (it may involve looping on the traffic controller lock), there is a means to avoid this notify call when in fact no process is waiting for the unlocking of the page table lock. This means is implemented by the cell sst.ptl_wait_ct. This cell is zeroed only by the notify code in the traffic controller when it notifies the PTL event, protected by the traffic controller lock. All code that unlocks the page table lock inspects this cell after unlocking it; if nonzero, it notifies the PTL event.

Any process that finds the page table lock locked on the fault side transfers to pxss$ptl_wait. This entry, once it locks the traffic controller lock, increments the cell sst.ptl_wait_ct. From this point on, any process that unlocks the page table lock must call the traffic controller to notify the PTL event. The page table lock is then inspected, under the protection of the traffic controller lock, to see if it has been unlocked since the fault side found it locked. If so, the process is made to wait for the PTL event, since it is guaranteed that sst.ptl_wait_ct is nonzero (as it is only zeroed under protection of the traffic controller lock, now held by this process), and thus, that the PTL event will be notified, even if the page table lock has been unlocked since the last check, for the process that unlocked it checks afterwards the contents of sst.ptl_wait_ct. On the other hand, if the page table lock is found to be unlocked at this second check, the cell sst.ptl_wait_ct is decremented by one, as this process will not wait, but retry. Thus, in this case, the process returns out of the traffic controller as if the PTL event had been notified, causing the page fault to be retried.

There are two code sequences in the system that unlock the page table lock. One is the subroutine unlock_ptl in page fault, and the other is the code in the traffic controller page control wait entries that unlocks the page table lock once the traffic controller lock is locked. The unlock_ptl subroutine checks sst.ptl_wait_ct and calls the traffic controller page-control event notification routine, via the stack-sharing mechanism described earlier in this section. Normally, this routine is only called from the interrupt side "done" code of page control; this is the point to which the traffic controller returns. The value of the event ID (the PTL event), which in this case can only be notified from the unlock_ptl code, causes the interrupt side routine to return to the unlock_ptl code. The return address, being the value of index register 7, is saved in pds$arg_3 during this call.

The code sequence in the traffic controller that unlocks the page table lock calls an internal traffic controller notification primitive ("n3") to notify the PTL event if sst.ptl_wait_ct was not zero after the page-table lock was unlocked.

FSDCT Paging

The segment FSDCT in ring zero contains all of the free-storage bit maps for all mounted physical volumes. This can grow to be quite large, and thus, this segment is a pageable segment, subject to demand paging behavior for its entire extent. The information in the header of this segment is used by volume management, where the pageability of this segment presents no problems. Similarly, all deposition of addresses (freeing of disk records by turning on bits in this segment) are done by segment control (the update_vtoce and truncate_vtoce functions), and some of the peripheral services of page control (e.g., pc$truncate_deposit_all). Again, pageability is no problem. The withdrawing of addresses, however, is performed by the page-reading function in ALM page control, invoked from the fault side and various functions on the call side (such as abs-wiring for I/O buffer usage).

The ALM page control kernel may not itself take page faults. However, a mechanism exists to allow the page-reading function to achieve paging-in of the FSDCT without taking a page fault. This mechanism relies on the multiplex wait protocol and the fault-side retry mechanism. More precisely, either the fault side or the call side, when made to wait for an event via the traffic controller, will retry the operation that caused them to wait for that event. In the case of the fault side, this means taking the original page fault over again. In the case of the call side, this means re-evaluating PTW states, and calling ALM entries to perform state transitions based upon these decisions. The essence of the mechanism is to initiate the read-in of the needed FSDCT page (when allocation is required) instead of the requested page, and causing the faulting process or the call side to wait for the event associated with this paging-in instead. When this read-in is finished (the event is notified), the page-read function will probably find the needed FSDCT page in main memory, and thus be able to proceed as though it were there to start with.

It is not ensured in any way that the FSDCT page paged in in such a manner will still be in main memory when the page-read function inspects it again. In this case, another read will be started for it, and the operation repeated. This is as deterministic as an ordinary page fault; it is not necessary that this operation complete in any given number of retries, but simply optimal to the behavior of the affected process. Similarly, the disk-record allocation function (free_store$withdraw) may progress through several pages of the FSDCT to find the necessary allocation. This will cause these pages to be paged in successively, with the faulting process or the call side being made to wait for each one in succession. Between the time that these processes are made to wait and the time that they retry the search through the actual FSDCT for a free record, other processes can deposit pages (paging in the FSDCT via normal paging) and withdraw record addresses (via the same mechanism). There is no interlock against this, or any need for one. The state of a given bi-map is recorded in the PVTE for that volume, and is not dependent upon any allocation that might be in progress.

The necessity of the read-page function to have the necessary pages of the FSDCT in main memory to complete its task is very much akin to the necessity of having a set of pages in main memory to initiate execution of multi-operand EIS instructions; nothing ensures that all the required pages will ever come into main memory, although every retry attempt tries to bring them there. Once they are all there, the operation proceeds. The process is effectively roadblocked until all these pages can actually be found in main memory at once; how long this is depends solely on system paging load.

The interface to the disk-record allocation function involves two error exits; one for the out-of-physical volume condition (no more records to allocate) and another for a needed page of FSDCT not being in main memory. In the latter case, the AST entry pointer and PTW pointer for the needed page are returned in the AQ register to the page-read function, which now redefines its task to be the reading-in of that page, including the allocation of a main memory frame and all other actions normally associated with the page-read function (see "Page Reading" later in this section). The last step of this function is to return a wait event to the caller. In this case, it will be that wait event associated with the FSDCT page.

Per-Process Trace List

(page trace)

Page control maintains in the PDS of each process a circular trace buffer of page readings, being mostly page-faults. The primary use of this trace is to drive the post-purge function at eligibility loss time (see "Post Purge" under "Services"). A secondary use is for the user commands "page_trace" and "cumulative_page_trace," which display and interpret this information. To the latter end, various other mechanisms in the system make entries in this trace list corresponding to such events as linkage faults, segment faults, and schedulings. The trace region is at the symbol pds$trace, in the wired part of the PDS. The format of this region is given in the include file page_trace.incl.pl1, as well as the format of the trace entries.

This trace list is maintained by the subroutines "page_util_enter" and "enter" in page_fault.

Disk Record Allocation/Deallocation

A bit map of unallocated records on every physical volume is kept in the segment FSDCT. The parameters that describe each bit map, including the offset of the bit map in the FSDCT itself and the state variables of the allocation/deallocation mechanism for that volume, are kept in the PVT entry for the volume concerned (see "Data Bases," Section VI).

The basic allocation strategy maintains a pointer into each map (pvte.curwd), that points to the last word in the map in which free records were found. Each word of map describes 32 records. When a request is made for a record, that word is scanned for another one-bit. Successfully finding a one-bit on causes the record defined by that bit to be returned (allocated), the bit being then turned off. The result of the floating-point normalization is checked by testing that the bit it claimed to be on is actually on; failure produces a "unprotected or reused address" crash.

Before any word of the FSDCT is inspected, a check is made (by inspecting the FSDCT's page table) that the necessary page of FSDCT is in main memory (see "FSDCT paging" earlier). Before any allocation is attempted, a check is made to see if there are any free records on the specific volume at all; if not, an error return is taken causing the ultimate invocation of the segment mover.

As each word is depleted of free storage bits, the next word in the bit map is moved to. The code that accomplishes this (in "withdraw" in free_store) contains the remains of an algorithm which used to interlace assignment over drives, prior to the advent of physical volumes. The effect of this in every case is to move on word-by-word up the bit map, and come around again to the beginning when the end has been reached. Thus, the pointer pvte.curwd cycles through the bit map for each drive.

Whenever one hundred deposits are made against a given drive, the deposit code resets this counter (pvte.relct) and resets the "curwd" pointer to zero. This has the effect of packing records tighter on each pack; whenever one hundred records have been deposited, the scan for the next free address is thus reset to the lowest address in the paging region of a pack.

The code for depositing (freeing) an address is trivial; the bit corresponding to that address is turned on. If already on, a reused-address error has occurred, indicating page control malfunction, and the system is crashed.

Any reused address detected by the program free_store caused the "vol_trouble" bit in the PVT entry for that volume to be turned on; this causes a volume salvage the next time that volume is accepted, even if ESD succeeds.

INTERNAL INTERFACES

This section explains the structure and function of the basic page-state manipulating subroutines of ALM page control. Some are externally accessible from PL/I page control via the transfer vector "page." Many are not; it is the functions provided by these interfaces in terms of which the Page Control Services of Section IX will be described. Figure 8-8 shows the call flow of most of these routines. Utility subroutines are described in the section following this.

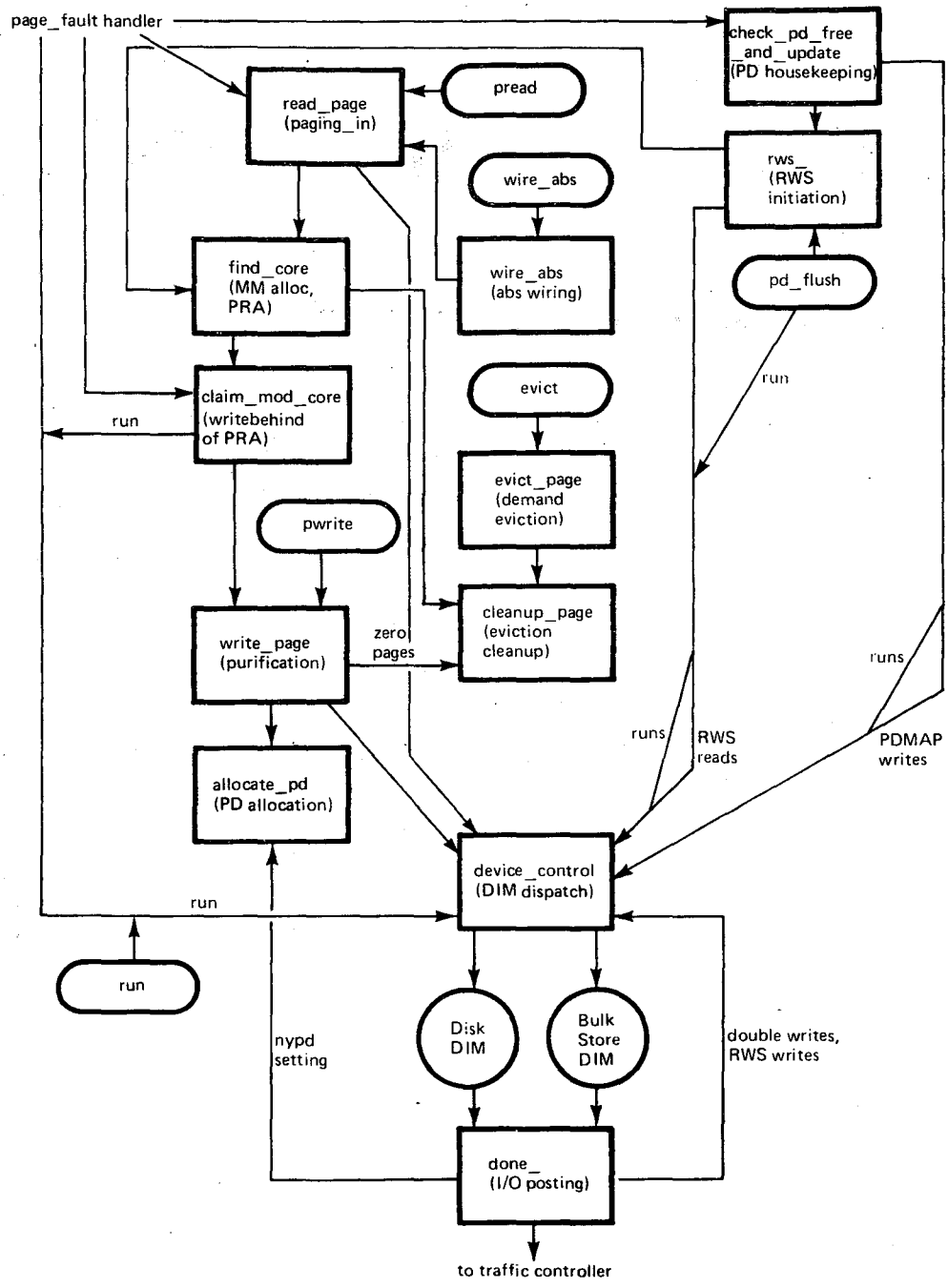


Figure 8-8. ALM Page Control Call Flow

Main Memory Frame Allocation

(find_core in page_fault)

Perhaps the most fundamental interface of all is that which finds a free main memory frame entry into which a page is read, including that performed on behalf of a page fault. This is the routine find_core, in the program page_fault. This routine is invoked by ALM page control whenever a frame of main memory is needed, other than some specific frame (abs-wiring).

The basic mechanism of allocating a main memory frame is the running of the main memory replacement algorithm, which runs exactly as described in Section V. If there are free frames available, the program that freed them moved them to the head of the "used list," and the pointer sst.usedp points to a free frame. If none are free, the used list is searched for a frame that contains a page that can be evicted without any I/O, that has not been recently used. Frames that do not meet these criteria are moved to behind the pointer. Wired and abs-wired frames are skipped too. If fifteen frames are passed over because of the fact that they would need I/O to evict their pages, claim_mod_core, the purifier of pages, is invoked, which starts those I/Os, and the scan continues. It may be so that claim_mod_core found some pages of zeros, or caused the DIMs to call the interrupt side, in either case putting claimable pages ahead of the used-list pointer. If a tremendous number of frames are rejected, the system is crashed with the message "out of core" (main memory).

When a frame is found which meets the criteria, an attempt is made to evict it. This attempt consists of turning off the bit ptw.df, which allows the hardware to use the page, clearing the associative memories of the system, and testing to see if the page was modified any time in the interval between the original decision that it was not modified (hence no I/O was necessary to evict it) and this clear of the associative memories. If it indeed was modified in this window, the eviction has failed, the access bit (ptw.df) is restored, and the search for an acceptable page continues. If it was not, the eviction is successful, and cleanup_page (see "Eviction Cleanup," below) is invoked to complete the eviction. The core map entry is left behind the used pointer (most recently used frame), with the field cme.ptwp being zero, this indicating the fact that it is free. The core map entry representing the frame made available is designated by the value of index register 4, on the return from find_core. Since find_core inspects many PTWs, and may call claim_mod_core, which may involve many PDMEs and CMEs and PTWs, no index registers are preserved by find_core.

The reader should note that pages that require updating to the paging device, even though they are not modified, require I/O to be evicted, and are thus not acceptable to find_core.

Replacement Algorithm Writebehind

(claim_mod_core in page_fault)

The main memory frame allocation function avoids frames containing pages that require I/O for their eviction so that it can return a usable page frame to its caller in minimal real time, allowing the read operation that the caller is sure to initiate to be started as soon as possible. This allows all writing on behalf of the replacement algorithm to be initiated while the read is in progress. This starting of writes is performed by the subroutine claim_mod_core in page_fault. This subroutine is invoked at the end of every page fault. When the main memory frame allocation function is invoked on behalf of some other action than a page fault, it is not invoked. In this case, the next page fault simply causes claim_mod_core to consider a larger set of pages than otherwise. The subroutine claim_mod_core is also invoked by find_core when fifteen frames have been skipped because of the need to perform I/O to accomplish their evictions.

Three functions are performed by claim_mod_core; any page frame skipped by find_core because of the need to do I/O to effect its eviction (whether actually modified or simply not yet on the paging device (nypd)) has such I/O started upon it. This is done via a call to write_page, the page writing/purifying function. Pages with their used bits on have them turned off; this is normally a function of the replacement algorithm, but the latter (find_core) must leave these bits on so that claim_mod_core will not initiate writes on pages that ought not to be evicted in the near future. A check is made to determine that no more than thirty writes are outstanding (page control disk writes only, not VTOCE writes), and the DIMs are "run" (see "DIM Interface and 'Running'" earlier) until this is so. This third function ensures that find_core is not processing vast numbers of frames because a very large number of writes have not completed.

The routine claim_mod_core processes all frames from the point it last left off (indicated by sst.wusedp) to the tail of the used list (where find_core is now, indicated by sst.usedp). Since calls to claim_mod_core might call the page-write function to start writes, and this might involve calling DIMs, which might call the interrupt side, the state of the pointer sst.usedp and the position of individual frames in the list maybe affected by invoking claim_mod_core. Also, claim_mod_core preserves no index registers.

Page Writing/Purification

(write_page in page_fault)

The contract of the page writing/purification function (the routine write_page in page_fault) is to start only I/O necessary to ensure that there is a good copy of a page outside of main memory (excepting the case where the page becomes modified after invocation of this routine). If some function, such as the deactivation-time service (pc$cleanup) wishes to purify the main memory page unconditionally, it must take steps that no process can reference the page (i.e., setfaulting all of the SDWs, as the deactivation function of segment control does). Purification consists of making the copy in main memory "pure," i.e., not modified with respect to secondary storage or paging device (whichever is appropriate).

The basic task of write-page is to initiate an I/O operation, writing the page out. The peripheral tasks consist of making all of the state transitions upon the PTW and CME of the page and page frame to set them 'out-of-service' (meaning "I/O in progress," not unusable), checking for all zeros, and checking whether allocation of a paging device record is in order.

The routine write_page is also used to cause the writing of unmodified pages (pure pages) which are not on the paging device (ptw.nypd) to the paging device. Thus, write_page performs precisely that function required by the replacement algorithm to make a page evictable without any I/O. It can also be seen that write_page is invoked on pages that are both modified and unmodified in main memory. The stack variable "mod_flag" tells what case is true (zero = not mod).

When invoked on a modified page (ptw.phm is on), write_page checks for a page of all zeros (unless the switch aste.gtpd is on to inhibit this). Such pages are evicted at this time, by write_page, calling cleanup_page to finish the eviction. As in find_core (see "Main Frame Allocation" above), a two-step trial eviction is necessary. When it has been determined that a page is all zeros, access is removed (ptw.df set off), the associative memories of the system cleared, and the page checked for zeros. If found not to be zero at this second check, it is treated as though it were not found as zero the first time. Any PD record associated with such pages are freed by a call to pd_util$pd_delete_. At this time, the disk record address of the page is nulled (see Section VII, "Address Management"), holding it for either later resurrection or deposition by segment control.

The routine write_page turns off the modified bit in the PTW, (ptw.phm) using a lock-type instruction (ANSA). The PTW associative memories of all processors are then cleared. If the page is modified before the clear of associative memories, but after write_page noted that the page was modified, the data bases will be modified to indicate that the page was modified (such as pdme.mod), and the write will proceed in any case. If any processor modifies the page after the clear of associative memories, the next attempt to evict the page will find that it was modified, and thus the copy written by this invocation of write_page was invalid. Note that access to a page remains on during a write.

If invoked on a modified page, the routine write_page turns on "file modified" switches (aste.fms) in the ASTE of this segment and all superior directories, unless aste.gtms is on, inhibiting this. (See the description of this flag in Section II.) The routine write_page also rethreads the PDME for a page which it writes (via a call to pd_util$rethread) to the tail (most recently used) position of the PD used list. This is to implement the part of the paging device management algorithm (see Section V) that states that pages in main memory are to be considered among the most recently used.

One very critical action of write_page is to check if the page being written must be allocated a paging device record; this check is made by the subroutine allocate_pd (described below) in all cases (other than a zero page). Whether or not the page is modified, allocate_pd allocates a PD record if it should, if it can, and one is not already allocated for this page.

The final action of write_page is to invoke device_control$write to actually call the appropriate DIM to start a page write. Since it is part of the DIM interface (see "DIM Interface" earlier in this section) that the request being issued may even be completed during that call, this must be the last action taken by write_page.

The routine write_page is invoked with index register 4 pointing to the CME of the frame that is to be written. It expects index 2 and pointer register 2 to describe the PTW of this page, and index 3 the ASTE of its segment. All of these registers will be preserved. No statement is made about the final state of the page or frame, or whether or not it will be out of service upon return from write_page.

The routine write_page is normally invoked from ALM page control, on behalf of claim_mod_core, with the registers set as above. However, it may also be invoked as page$pwrite from call-side PL/I code. In this case, the interface routine page_fault$pwrite is invoked, which establishes the ALM page control environment, and the necessary pointers and index registers, and calls the routine pwrite.

Page Reading

(paging-in) function - (read_page and read_page_abs in page_fault)

The basic task of the reading function is to bring a page of a segment into main memory; if null or nulled, a page of zeros will be created. Generally, a read of disk or paging device will be required, and the page-reading function will initiate this read. The page reading function indicates to its caller whether or not waiting will be required by this caller.

It is part of the task of the page-reading function to check that both adequate record quota and adequate disk storage space are available to accommodate the page. Quota must be checked each time a page with a null or nulled address must be paged in for it is at that time quota is charged. Physical device allocation must be checked each time a page with a null address is paged in. If either of these operations cannot be successfully performed, i.e., adequate allocations do not exist, action can be taken before the page is created in main memory. It is only legal for the fault side to encounter out-of-quota or out-of-physical volume situations; all segments treated by the call side should be quota-inhibited and prewithdrawn.

The page-reading function has two entries, read_page and read_page_abs, in the module "page_fault." The usual entry is read_page, which, as part of its task, locates a main memory frame into which to read the requested page, via a call to the main memory frame allocation function find_core. The other entry, used only by the abs_wiring function, is supplied the identity of a specific main memory frame into which the paging-in is to be done. Either entry expects to be called, via TSX7 instruction, with index register 2 set to the relative address of the PTW of the page to be read in, and index 3 to the relative address of its AST entry in the SST. The routines return with not only these registers set, but index 4 set to the relative address of the core map entry of the main memory frame into which the paging-in was done. The routines return to the location past the TSX7 instruction if they started I/O that has not been completed, in which case the upper A register has the event ID to wait for. Otherwise, if no incomplete I/O exists, or none was started at all, a return to the location two locations beyond the TSX7 is executed.

If the page-reading function encounters a page on which a read-write sequence (RWS) is in progress, the caller is returned the event ID of that RWS. This will cause the call side to ultimately set a notify-requested bit in the affected PDME. The fault side will initiate an RWS abort (turning on pdme.abort) in this case.

Other actions of the page-reading function include maintaining the current-length and records-used ASTE parameters of the segment in the case where a page is created (zero page paged in), and performing the CME and PTW state transitions associated with setting a page out of service in other cases. When the page being paged in is on the paging device, the associated PDME is rethreaded to the tail of the PD used list, in keeping with the policy that all pages in main memory are among the most recently used on the paging device. As is the case with the page-writing function, the actual call to device_control (in this case device_control$read), the DIM dispatcher, must be the last action taken, for the page may not even be out of service on return from this call.

A major consideration of the page-reading function is to loop, redefining its arguments, when the call to the disk-record allocator (free_store$withdraw) indicates that a page of the FSDCT must be paged in to perform the allocation. As explained under "FSDCT Paging" earlier, the page reading function must redirect itself to page in a page of the FSDCT instead of the page passed as an argument, when paging in that page involves allocating a disk record, and that allocation requires paging in the FSDCT. In this case, whatever wait event or lack thereof results from such activity will be returned to the caller of read_page (or read_page_abs) to wait on.

The page-reading function is normally invoked at the read_page entry point, via the routine "readin" in the page-fault handler (see "Page Fault Handling" in Section IX). However, it may also be invoked from page$pread from call-side PL/I code, such as the process-loading function. In this case, the interface routine page_fault$pread is invoked, which establishes the ALM page control environment, and calls read_page. This interface routine conveys the wait-event ID returned by read_page to its caller, returning zero if there was none. However, in the case of bulk store I/O, the interface routine page_fault$pread "runs" the bulk store DIM in a loop to await the completion of the I/O. This is to obviate the need for a separate bulk store waiting mechanism for the process-loading function. Thus, only disk I/O or RWS events are returned to the caller of page$pread.

Paging Device Record Allocator

(allocate_pd in page_fault)

The paging device record allocator is invoked at two times; at the completion of a disk read, and during the page-writing function. Its task is to determine whether a page is or should be on the paging device. If the latter is the case, either the bit ptw.nypd is set (if invoked on behalf of a disk-read completion) or the page is actually migrated to the paging device (if invoked from the page-writing function).

Migrating a page to the paging device consists of finding a free paging device record, and updating the CME and PTW associated with the page being migrated, as well as the PDME for the free record found, to indicate that they are all associated with the same page. The routine allocate_pd performs an alternate return depending on whether or not it migrated the page to the paging device as a result of this invocation.

The decision as to whether a page should go on the paging device involves the decisions as to whether it is already there, whether the segment to which it belongs has the "global transparent to paging device (gtpd)" attribute, explicitly inhibiting this action, whether or not there is actually an enabled paging device in use, and the consideration of the ptw.first usage-optimizing feature (see the description of this bit in the PTW breakdown in Section VI.

When invoked on behalf of the completion of a disk interrupt, the page is not actually migrated to the paging device unless the "pd_writeahead" switch is set in the SST (sst.pd_writehead; see the description of this field in the SST breakdown in Section VI. This feature is not currently operative.) Rather, the bit ptw.nypd in the PTW is turned on. This bit tells the main-memory replacement algorithm that the page-writing function must be invoked to evict this page, allowing it to skip that page in a search for the "most available" page to evict. When the page-writing function is called for this page, on account of this bit, it will cause the paging device record allocator to be invoked once more at which time the page will actually be migrated to the paging device.

When the paging device record allocator actually decides to migrate a page to the paging device, there should be free records available on the paging device. The paging device management algorithm attempts to keep a free pool by ensuring the existence of a small fixed number free or being freed at the beginning of the processing of each page fault. Thus, the free paging device record at the head of the paging device used list is normally allocated to the page on behalf of which the paging device record allocator is being invoked. If the record at the head of the paging device used list is not free, an action known as a "PD Desperation" is performed. This action, performed by the PD Desperator, pd_util$force_get_pd, consists of walking down the PD used list no more than fifteen steps to find a paging device record whose page is evictable without a read-write sequence, or an eviction from main memory. A read-write sequence (RWS) may not be performed at this time; the call history of the paging device record allocator may well include the main memory frame allocation function, which is necessary to initiate an RWS, and is not recursive. Furthermore, the completion of the RWS could not be awaited at this time; ALM page control does not wait, but indicates wait events to its caller.

If a PD desperation fails, the paging device record allocator fails (see the SST breakdown in Section VI for the names and meanings of meters of this event), and the page is not migrated to the paging device. This causes the page-writing function to turn off any "nypd" PTW bit which may be on, causing all attempts to migrate the page to the paging device for this activation to be abandoned.

The paging device record allocator expects to be called via a TSX7 with pointer register 2 and index register 2 describing the PTW of a page for which paging device allocation must be checked and/or performed. Index register 3 must point to the AST entry of the segment containing the page. The page must be in main memory, and not out of service or undergoing a read-write sequence (RWS), and index register 4 must describe the core map entry (CME) for the main memory frame it occupies. The stack variables "devadd" and "did" must contain the record address (in the format described at the beginning of Section VI) and the PVT index for the page.

The paging device record allocator returns to the location beyond the TSX7 if it did not allocate the page to the paging device as a result of this invocation, and two locations beyond if it did. If it migrated the page to the paging device, the stack variables "devadd" and "did" will be modified to reflect the paging device record address of the page, as well as the core map entry of the page.

RWS Initiator

(rws_ in pd_util)

The RWS initiator is supplied the identification of a paging device record (as a relative pointer to its PDME) and is responsible for starting a read-write sequence (RWS) on that PD record. It invokes the main-memory frame allocator (find_core in page-fault) to allocate a frame for the RWS, and the DIM dispatcher device_control$read to start the read cycle of the RWS. It threads the CME of the main memory frame and the PDME of the paging device record out of the main memory and paging device used lists, respectively, and performs the necessary state transitions upon all of these objects to indicate that the read cycle of an RWS is under way. The RWS initiator neither awaits completion of the read cycle nor initiates the write cycle; the former is done by either the PD replacement function or the interface routine pd_util$pd_flush, the latter is done by the interrupt side.

The RWS initiator never allows more than thirty RWSs to be outstanding; when it has initiated the thirty-first RWS, it "runs" the DIMs until one of them

has competed (i.e., the count sst.pd_wtct has gone down).

As with the page-reading and page-writing functions, the call to device_control to actually start the reading I/O is the last action performed by the RWS initiator, as the RWS it initiates could be over (especially on account of error) by time the return from this call is complete.

The RWS initiator is called via a TSX7 instruction from ALM page control. It expects index register 1 to point to the PDME for the PD record to undergo RWS. It saves no registers, it has no alternate returns. It destroys the contents of the stack variable "ptp_astep," used by the page-reading function, among others.

The RWS initiator is used by the PD replacement function and a large number of call-side functions, such as the deactivation-time service and the PD reconfiguration function. In these latter cases it is called as page$pd_flush from PL/I code, which invokes the interface routine pd_util$pd_flush. This interface routine establishes the ALM page control environment, and invokes the RWS initiator upon the PDME located by the PL/I pointer argument to this routine. The interface routine also awaits the completion of the read cycle of the RWS initiated, by "running" the bulk store DIM. This maintains the convention that no RWS read cycles may be in progress at the time the page table lock is unlocked.

Paging Device Housekeeping and Replacement

(check_pd_free_and_update in pd_util)

This function serves to keep a small pool of free paging device records available for the paging device record allocator at all times. The paging device record allocator cannot free records on demand except in certain special cases. The paging device housekeeping function also serves to write out the paging device map to the first few records of the bulk store every second. This copy is maintained for the use of the post-crash PD flush (see description of that in Section IX).

The paging device housekeeping function is invoked at the beginning of the processing of every page fault, from the page-fault handler. It initiates the writing of the paging device map if that has not been done within the last second; this map writing is done with the "no_interrupt" flag to the DIM set on. The interrupt side of page control does not want to be informed when this I/O has completed.

The PD housekeeping function implements the paging device replacement algorithm outlined in Section V. The paging device used list is scanned from least-recently-seen-used to most-recently-seen-used end until ten records are free or in the process of undergoing RWS. An RWS is initiated for each PD record passed which is modified with respect to disk; the RWS initiator just described is used. Each record inspected that is not modified with respect to disk is freed; its page contents are migrated off the paging device by modifying the PTW of that page. This PTW currently describes this PD record. It is made to describe the disk record currently described in the PDME. Pages that are found, by inspection of the PTW designated by the PDME field pdme.ptwp, to be in main memory, cause their PDMEs not to be claimed, but rather, made to be "recently seen as used" by rethreading them to the tail of the PD used list. This is in keeping with the policy that those PD records seen in main memory are to be considered among the most recently used.

The final action of PD housekeeping is to check that no RWS read cycles are in progress. RWS read cycles may not be in progress when the page table lock is unlocked, and it is the responsibility of whichever agency invokes the RWS initiator to see that they complete before it exists. The strategy of waiting for the (bulk store) reads to complete all at once allows the RWS initiator to start all of these reads in parallel. This overlap optimizes performance via the queueing facility of the bulk store DIM. The PD housekeeping function "runs" the DIMs in a loop until no more RWS read cycles (counted by sst.rws_reads_os) are outstanding. During this looping, the bulk store DIM will invoke the interrupt side to initiate the RWS write cycles.

The PD housekeeping function is invoked via a TSX7 instruction from ALM page control. It preserves no registers, and has no alternate returns.

Eviction Cleanup

(cleanup_page in page fault, and code in pc$cleanup and pc$truncate)

The eviction cleanup function consists of modifying all page control data bases necessary to indicate that a page has been evicted from main memory. This function does not include the actual turning-off of the PTW access bit, ptw.df. The latter involves associative-memory and cache clearing, and turning it back on if the page was found to be modified after the clear had taken effect. It is the responsibility of the eviction cleanup function to modify all other data objects once access to a page has successfully been turned off. In ALM page control, this function is performed by the subroutine cleanup_page in page_fault. This routine is invoked by the main-memory replacement algorithm, when it evicts a page, and by the demand-eviction and abs.wiring functions (see description later in this section) when evictions are performed.

The call side also evicts pages, in the routine pc$cleanup invoked on behalf of the segment control deactivation function, and in the truncation function. PL/I code in these routines performs work similar to that of cleanup_page. This work is much simpler in the case of truncation.

Eviction cleanup consists of maintaining the AST entry of the segment and freeing the main memory page from which the page was evicted. The number of pages in main memory is updated. If the page evicted contained zeros (i.e., its address is now nulled, the "number of records used" of the segment must be adjusted, as well as the record quota account against which the segment's pages are charged. If no more pages of the segment are in main memory after this eviction, the ASTE "init" bit (used by the AST replacement algorithm) must be turned on. If the highest-addressed page of the segment which is nonnulled/null or was in main memory was evicted, the current length of the segment is adjusted. The disk or PD address from the core map entry of the frame from which the eviction is being performed is placed back in the PTW for the page. The PTW "first" bit (for the optimizing algorithm described under the description of sst.ptw_first in Section VI is turned off, indicating that the page has been evicted at least once from main memory since activation.

The routine cleanup_page is invoked via a TSX7 instruction from ALM page control. It expects pointer register 2 and index register 2 to describe the PTW of the page being evicted, and index 4 to describe the CME of the main memory frame from which it is being evicted. It will preserve these registers, as well as set index 3 to the ASTE. There are no alternate returns.

Per-Page Cache Management

(cam_cache in page_fault)

The general strategies for managing the Multics Processor caches are described under "Encacheability Control" in Section II. These strategies cover modification of main memory by several processors, and by all I/O devices except I/O devices used for paging. The per-page cache management strategy covers these latter cases; when a page is read in from paging device or disk, the contents of main memory locations which may be in processor caches will be modified without changing the contents of these cache locations. The avoidance of this situation is the goal of the per-page cache management strategy.

Paging I/O has the unique property that a main memory frame into which a page of a segment is being read is guaranteed to contain no information that any processor (or process) can access. Therefore, if it could be ensured that no words of that page appeared in any processor's cache at the time the read was begun, there would be no chance that the reading in of data by the IOM could contradict any data in a processor cache. Thus, it would be adequate to clear the caches of all processors at the time a page of a segment was evicted from main memory, i.e., made inaccessible to the processors of the system.

The Multics processor cache includes a feature known as "selective clear," a hardware mechanism for iterating through all the columns and blocks of the cache, and invalidating the contents of any block that contains information from a given page. This mechanism is available via the CAMP instruction, with the "4,du" bit on in its effective (internal) address. The frame is identified by the upper bits of this address. This instruction also clears all processor PTW associative memories, which is desired at page eviction time. Thus, at page eviction time, all system processors are forced to execute an instance of this instruction to clear all words of the page being evicted out of their caches, and all PTWs out of their PTW associative memories. The instance of this instruction so constructed is stored in scs$cam_pair; the general CAM/connect strategy is described in the Multics Reconfiguration PLM, Order No. AN71.

The function available to ALM page control as the "cam_cache" routine is also available to PL/I code as page$cam_cache, which invokes the interface routine cam_cache_ext in page_fault. However, most PL/I code calls page$cam before unlocking the page table lock, which clears all system caches and associative memories totally.

It is critical to this strategy that the abs_segs used by page control to check page frames for zeros not be encacheable.

Demand Eviction

(evict_page$evict_page)

The demand page eviction function is one called by the main-memory deconfiguration function (see Section IX), on behalf of the system reconfiguration software, and on behalf of the I/O buffer abs-wiring function. In the latter case, it is used to evict the previous resident of a main-memory frame into which a page of an I/O buffer segment is going to be abs-wired. It is also the responsibility of the demand eviction function to inform its caller of any RWS or page transfer I/O in progress in the main memory frame being vacated (having a page evicted from it).

The demand page eviction function is called as page$evict_page from PL/I code only. It is called with a PL/I pointer to the core map entry representing the main memory frame from which it is to be vacated. It returns a wait event ID; if that is zero, it has successfully vacated the frame, if not, the caller must await that event (via the multiplex wait protocol and the call-side wait-coordinator) and call evict_page again when the event has happened. The demand page eviction function is an excellent example of those functions that perform successive state transitions upon page control objects, which must be constrained from retrogression via the setting of control bits. In this case, the caller of the demand-eviction function must have set either of the CME bits cme.removing or cme.abs_w, to ensure the success of the vacating (prevent the main memory frame allocator from allocating the frame).

The demand page eviction function begins by checking that no RWS or ordinary page-transfer (ptw.os on) I/O is in progress in the frame being vacated; if so, the caller is returned the event ID corresponding to the operation in progress. If, via the multiplex wait protocol, the caller chooses to wait for this event via the call-side wait coordinator page$pwait, the latter will turn on the appropriate notify-requested bits to cause the interrupt side to notify the completion of these events. If there is no I/O going on in that frame, the demand page eviction function will successfully complete in this call, i.e., there will be no waiting.

If the page in the main memory frame is wired (but may <u>not</u> be abs_wired), it must be moved to another main memory frame, in such a way that it is never made inaccessible to the system processors. Since the page may be being modified by the system processors there is no way to move the page while other processors are accessing it. Furthermore, it is not desirable to change the contents of the PTW, which will be necessary, while other processors are using it. Thus, a mechanism is provided to halt all of the system processors except the one executing this code, until this processor releases them. This service is provided by the CAM/connect mechanism, which sets appropriate flags in the SCS segment when this is the case. First, the main memory frame allocator (find_core) is invoked to obtain a page frame into which to move the wired page. The state of the "modified" bit (ptw.phm) of the page being moved is saved, and it is turned off. This must be done in one unitary (key-line) operation, lest a modification between the inspection and the turning-off be lost. All of the system processors, except the one executing, are then stopped, and all PTW associative memories cleared, via a call to cam_with_wait in page_fault. This routine also causes all words of the old frame to be selectively cleared out of all the caches of the system processors. The contents of the old frame are then moved to the new frame via the use of two non-encacheable abs-segs. If, after so doing, the PTW "modified" bit (ptw.phm) has not come on (since it was turned off) the contents of the old frame and new frame are the same. If not, the contents are moved once again (this is metered by sst.recopies). The contents cannot now possibly change, since all processors are halted. The possible modification just noted is then "or'ed" into the PTW "modified" bit (ptw.phm), and the PTW main memory address (ptw.add) is changed to describe the new frame. The system processors are then released via zeroing the cell scs$cam_wait, on which they all are looping. The CME for the old frame is made to be free (although it has one of the bits cme.abs_w or cme.removing protecting against its accidental claiming), and the CME for the new frame is made to describe the page moved, which had been described by the CME for the old frame.

If the page in the frame being vacated is not wired, then the task is vastly simplified, as it is permissible to make the page inaccessible. This is precisely what is done. The PTW access bit, ptw.df, is turned off, and the system PTW associative memories are cleared, and the caches selectively cleared, as for any eviction. If the modified bit is not on after this clear, (it could not have been on before access was removed, or it would still be on), then a successful eviction has just been performed. The eviction cleanup function (page_fault$cleanup_page) is invoked to complete the details of the eviction, and the frame has been successfully vacated. If, on the other hand, the page was modified, either before access was turned off or after, we must move its contents to another frame, which is cheaper and faster than starting an I/O and causing the caller to wait for it. It is also deterministic; there is no

telling how many times the page could be modified while the caller waited for it, while moving the page avoids the entire issue. Thus, the main memory frame allocation function (find_core) is invoked to obtain a frame, and the page is moved. System processors do not have to be halted, as opposed to the wired case, as the page was just made inaccessible, and the associative-memory clear mechanism ensures that no processors are left accessing it. The contents of the PTW address field (ptw.add) is changed to describe the new frame. The CME for the old frame is made free (although it is protected by cme.removing or cme.abs_w) and the CME for the new frame is made to describe the page moved. The access in the PTW is restored, i.e., ptw.phm is turned back on, and processors continue to use the page in its new location (although, however, if they attempt to access it before this, but after the time that access was revoked, such processors caused their processes to take a page fault, and wait for the page table lock, now held by this process). Again, the eviction is complete with no waiting.

Page abs-wiring

(evict_page$wire_abs)

The page abs-wiring function is used only by the segment abs-wiring service (in pc_contig, described in Section IX. It is invoked from PL/I page control only, given a page (as an ASTE pointer and page number), and a free CME (usually vacated via the demand-eviction function just described) into which to abs-wire the page. It assumes that the caller has set the bit cme.abs_w in the CME for the frame participating in the abs-wiring, to prevent any page other than the one being processed from coming into that frame. If the "wired" bit of the PTW for that page is not on, the abs-wiring function turns it on, indicating that the page, wherever it might be now, or wherever it may come, may not be evicted.

The abs-wiring function is among that class of ALM page control functions that either complete their task when called, or return a wait event on which the caller must wait, and call that primitive back when the event has happened. The abs-wiring function thus returns a zero wait event ID if it has unsuccessfully placed the page in the frame into which it is to be abs-wired, or an event ID on which to wait.

The first check made by the abs-wiring function is that the page is not already involved in I/O (ptw.os on). If so, the caller is made to wait for the completion of that I/O. Since ptw.wired has been turned on, a page out of service on a read will not be evicted once it comes in, and a page out of service on a write will be selected for no further writes by the main memory replacement algorithm.

If there is no I/O on the page in progress, two different cases occur for the cases of the page being in main memory already (may even be in the required frame already, via previous calls), or not in main memory. If it is in main memory, and in the required frame, the task is complete, and a zero event ID is returned to the caller. If not, the page must be moved from its current frame to the new frame. The task is identically that of the code in the demand page-eviction function which moves wired pages, as this page is now wired. This code is used, the page is moved (including halting all processors, etc.), and the abs-wiring is complete. If the page is not in main memory, the page-reading function (read-page-abs) is invoked to read the page in. As described under the description of page reading function (earlier) the caller of that function (and thus, in this case, the caller of the abs-wiring function) is made to wait for the completion of FSDCT paging I/O, RWS completion, or page-reading, whatever may be the case, and retrying through repeated calls. Thus, this path through the abs-wiring function uses the multiplex wait protocol of the caller to drive the FSDCT paging mechanism and await RWS completion transparently to the mechanism of the abs-wiring function.

Note that the version of the page-reading function used by the abs-wiring function (read_page_abs) does not allocate a main memory frame, but uses the specific main memory frame supplied by the caller, in this case, evict_page$wire_abs.

I/O Posting

(the Interrupt Side) (page_fault$done_)

The function of posting (processing the completion) of paging I/O operations is one of the most critical in page control. It includes performing all of the state transitions out of I/O or RWS states of page control data objects, all error processing, and the initiation of RWS write cycles and double-writes, and calling the traffic controller notify primitive.

The I/O posting function is implemented in the routine done in the module page-fault. It is invoked solely from the storage system DIMs when they notice I/O completions. It is accessed directly via TSX7 from the bulk store DIM, and via the interface page$done, which leads to the interface routine page_fault$done, by the disk DIM. The routine done_ is invoked in the ALM page control environment with the page table lock locked. Its parameters are the stack variables "core_add" and "errcode," containing the main memory address and status of the I/O operation which was completed.

One critical function of the interrupt side is to resurrect disk addresses upon the successful completion of RWSs, double writes, or other disk writes. As described in Section VII, this resurrection signifies just this successful disk writing, indicating that the address may safely be reported by pc$get_file_map to a VTOCE.

The interrupt side begins its task by looking at the core map entry indicated by the main memory address passed to it. It must either indicate an RWS in progress, or designate a PTW which has its out-of-service (ptw.os) bit on. It may designate a free main memory frame. We consider first the case of normal paging (non-RWS) I/O completion.

If a page read completes, a check is made to see if it completed unsuccessfully (with an error). The error actions, here as elsewhere, are described in "Error Strategy" earlier in this section. Assuming there was no error, the CME for the page frame of main memory is threaded back into the used list, as "most recently used." The out-of-service bit is turned off. The paging device record allocator is invoked to cause the "not-yet-on-paging-device" bit to be set if necessary. If the notify-requested flag was on in the CME, the special traffic controller interface pxss$page_notify (see "Traffic Controller Interface" earlier in this section) is invoked to notify the completion of the read.

In the case of a page write completion, with no error, the CME is threaded back into the used list as "least recently used" (best candidate for eviction), and the out-of-service bit in the PTW (ptw.os) turned off. If the write was for a page not on the paging device, the disk address is resurrected (made live, not nulled), and the bit aste.fmchanged turned on to trigger a VTOCE update. If it was a write for a page on the paging device, the PDME is inspected (pdme.double_writing) to see if it was a double-write (write to disk for a page on the paging device). Otherwise, it was a write to the paging device. If it was a write to the paging device, a check is made to see if a double-write should be initiated, based upon the properties of the page, the segment, and the double-writing control switch (sst.double_write) in the SST. (See the description of that switch in Section VI for the various decisions and interpretations.) if a double- write is to be started, the page is put back

out-of-service, the threading-in of the core map entry avoided, and a call made to the DIM dispatcher device_control$write to start the I/O. The bit pdme.double_writing is put on before this call is made to indicate to the interrupt side what action should be taken at the completion of that I/O. If double-writing is not to be started, the CME is threaded as stated, and the traffic controller called to notify the write-completion if required. If a double_write's completion was noted, the double_writing PDME flag (pdme.double_writing) is turned off, the disk address resurrected, and the PDME marked as not modified with respect to disk. Again, an optional notify follows. These actions, as the rest of the interrupt side posting logic (other than error handling) are shown in Figures 8-9 and 8-10.

The actions taken for the completion of RWS I/O depend on whether it is a read or write cycle that has completed. When a read cycle has completed (cme.io tells which), the write cycle is started by setting the bit cme.io, and starting the I/O for the write cycle. The CME and PDME involved remain out of their respective lists, and no notifications are performed.

The completion of the write cycle is more complex, as it implies the completion of the entire RWS. At the start, notification of the RWS event is performed via the special traffic controller entry pxss$rws_notify if any of the bits pdme.notify_requested, pdme.removing, or pdme.abort are set; any of these bits implies that some process is waiting for the RWS event. Assuming no error, (see "Error Strategy" for discussion of the error path here), the disk address in the PDME is resurrected, indicating successful transfer of the data to disk. If no abort (page fault by a process while the RWS was in progress) was observed (pdme.abort would have been set on by the fault side), the PTW for the page which underwent RWS is located from the PDME, and changed to contain the disk address from the PDME (it now contains a paging device address). The PDME is zeroed, and marked as free, being put into the PD used list. The main memory frame is similarly freed, being put into the main memory used list. If however, a post-crash PD flush was responsible for initiating the RWS (pdme.flushing on), the PTW (none exists) is not adjusted, nor is the PDME cleared or freed. Rather, the PDME flushing, RWS, and modified flags are turned off. This leaves the PDME intact for the call side to inspect, so that an error during the RWS can be determined to have happened or not by inspecting the nulled/live status of the disk address (pdme.devadd) in the PDME.

If an RWS abort was noticed, the main memory frame in which the RWS occurred is converted into a normal page-holding frame. The ASTE of the relevant segment is adjusted to indicate the proper number of pages in main memory, etc.; and the CME pointers are set to describe the PTW and ASTE. The RWS flags in the CME and the PDME are turned off. The "modified" status of the PDME, which has never been turned off, remains in effect. The PDME is put back in the PD used list. The CME is put back in the main memory used list, in most-recently-used position. The process which had turned on the abort bit, causing the abort, has already been notified, and is now either "ready" or waiting for the page table lock.

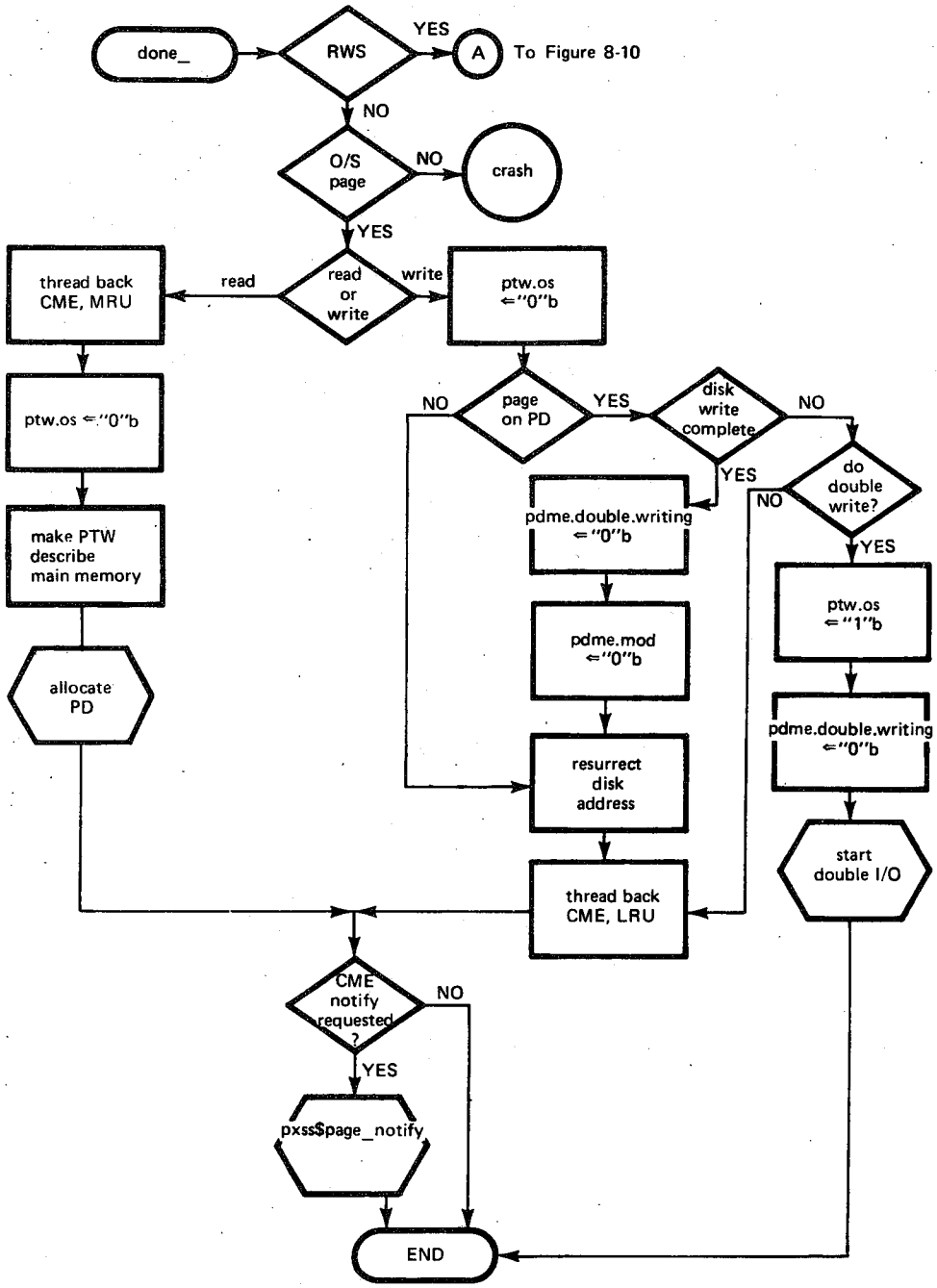


Figure 8-9. Page Control Interrupt Side, normal posting

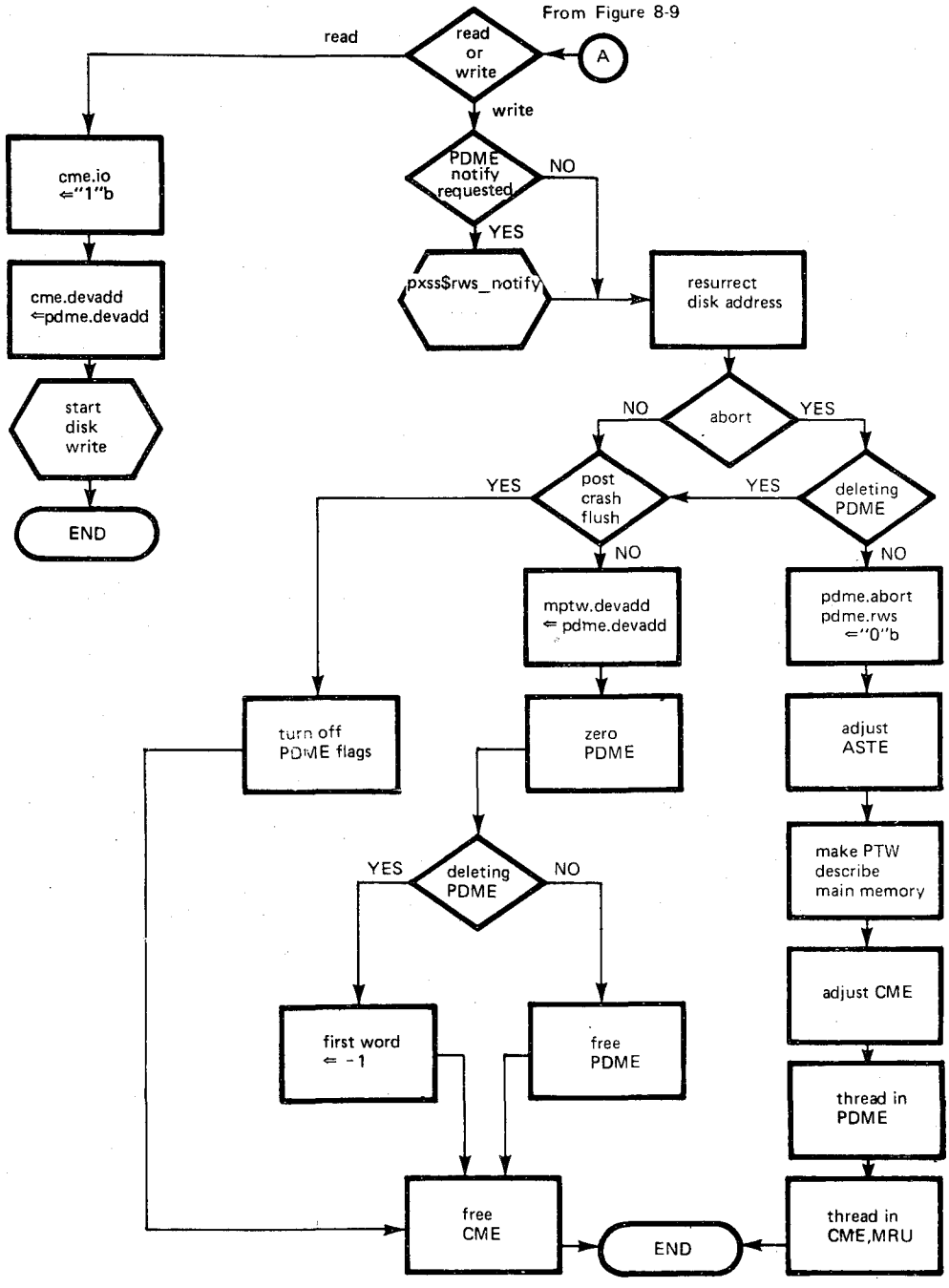


Figure 8-10. Page control Interrupt Side, RWS posting

Utility Subroutines

This discussion provides brief descriptions of the utility subroutines in ALM page control. All of these subroutines are in the modules page_fault and pd_util. A utility subroutine, by this definition, is a routine that does not affect the state of page control objects; PTWs, CMEs, PDMEs, other than perhaps rethreading them. Any routine that performs state transitions is among the critical agents described under "Internal Interfaces" earlier in this section. The name, function, and calling sequence of each of these routines, with whatever comments are appropriate, is given.

In page_fault

init_savex
: Called by TSX6, sets up page control index-seven save stack. Used to establish the ALM page control environment.

savex
: Called by TSX6, saves index 7 in the index-seven save stack. Any routine that saves index 7 via this routine transfers to "unsavex" to return.

unsavex
: Invoked via TRA, returns from a routine which called "savex" at its start. Pops the index-seven save stack.

thread_to_lru
: Invoked via TSX7, with index 4 pointing at a core map entry. Given a CME in the used list, rethreads it to the head (least recently used) position, adjusting whatever global page control pointers are necessary.

thread_out
: Invoked via TSX7, with index 4 pointing at a core map entry in the used list. Threads it out of the used list, zeroing its thread word, and changing whatever global page control pointers are necessary.

thread_in
: Invoked via TSX7, with index 4 pointing at a core map entry <u>not</u> in the used list. Threads it into the used list, and the head (least-recently used), updating global page control pointers.

thread_in_mru
: Invoked via TSX7, identical to thread_in, but the CME is placed at the <u>tail</u> (most-recently-used) end of the main memory used list.

thread_lru_ext
: Is a PL/I callable interface to thread_to_lru.

set_up_abs_seg
: Called via TSX6, with the main memory address of a page frame of main memory in the stack variable "core_add." Places a non-encacheable SDW for that page frame in the SDW slot for the segment "abs_seg1," and makes pointer register 0 (ap) point to it. This is used for checking for zeros and zeroing main memory frames on behalf of the page-writing and page-reading functions, respectively.

clear_ core
: called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Fills the frame with zeros, on behalf of the page-reading function.

check_for_zero
: Called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Sets the "zero" indicator register to indicate whether or not that frame of main memory contains all zeros, on behalf of the page-writing/purification function (see "Page Writing Function" earlier).

cam_cache
: Called via TSX7, with the main memory address of a page frame of main memory in the stack variable "core_add." Clears the PTW associative memories of all processors, and selectively clears the words of that main memory page out of their caches. Destroys index registers 0 and 1. Available to PL/I code as page$cam_cache.

cam_ptws

Called via TSX7, clears the PTW associative memories of all processors. Destroys index registers 0 and 1.

cam

Called via TSX7, clears PTW and SDW associative memories and all caches of all processors. Destroys index registers 0 and 1. Available to PL/I code as page$cam.

cam_with_wait

Called via TSX7 with the main memory address of a page frame of main memory in the stack variable "core_add." Clears the PTW associative memories of all processors, and selectively clears their caches of all words of that main memory frame. Furthermore, all processors except the executing processor are made to halt until the variable scs$cam_wait is zeroed by the executing processor. Destroys index registers 0 and 1.

reset_mode_reg

Called via TSX7 by the page fault handler, restarts the history registers and re-enables the cache after a (page) fault.

get_pvtx

Called via TSX7, with index register 3 pointing to an ASTE. Extracts the PVT index for that segment out of its ASTE, placing it in the stack variable "did" and in the accumulator.

get_astep_given_pdmep

Called via TSX6, with index 1 pointing at a paging device map entry (PDME). Determines the AST entry offset of the segment to which the page in the PD record described by the PDME belongs. Places it in index register 3, conventional register for ASTEs. May not be used during post-crash PD flush.

check_quota

Called via TSX7, with index 2 and pointer register 2 describing a PTW, and index 3 its ASTE, by the page-reading function. Returns if the page that is to be read in is not null and not nulled, or there exists a record of quota to support its being read in. Otherwise, transfers to "errquit" in the page fault handler to signal record quota overflow.

reset_quota

Called via TSX7, with index 3 pointing at the ASTE of a segment for which a page is being destroyed. Used by eviction cleanup to decrement "records used" for the quota account of some segment. The entry quotaw$cu_for_pc performs the same function for PL/I code in the program pc.

bump_quota

Called via TSX7 with index 3 pointing at the ASTE of a segment against which another page of record quota will be charged. Called by the page-reading function once it has determined that sufficient quota and disk space exist to create the page.

type_terminal_quota

Called via TSX6 from check_quota, bump_quota, and reset_quota, determines whether directory or segment page quota is involved, and whether the segment involved has quota checking suppressed (aste.nqsw on).

update_csl

Called via TSX7 when a page is found to be zero by the page-writing function. Recomputes the current length of the segment in the ASTE (aste.csl) by scanning the page table backwards.

In pd_util

pd_delete_

Called via TSX7; with index 1 pointing at a PDME, in the used list. The PDME is cleared and threaded to the head of the PD used list. It is assumed that the caller has evicted whatever page was in that record.

get_devadd

Called via TSX7, with index 1 pointing at a PDME. Creates a standard-format (see beginning of Section VI) paging-device record address for the PD record described by that PDME, returning it in the accumulator and the stack variable "devadd."

get_pdmep

Called via TSX7, with pointer register 2 describing a PTW. If this PTW describes a page that has a paging device record associated with it, returns two locations after the TSX7, with the relative offset of the PDME for the associated PD record in index 1 (conventional for PDMEs) and the upper half of the stack variable "pdmep." If not, returns <u>indirectly</u> (TRA 0,7*) through the first location after the TSX7, with index 1 and the upper half of the variable "pdmep" containing a -1.

pd_rethread

Called via TSX7, with index 1 pointing to a PDME in the PD used list. Threads that PDME to the tail (most recently used) position of the used list. If the switch sst.count_pdmes is on (see its description in Section VI), an esoteric form of metering is performed, recording in a histogram its distance down the list before rethreading.

pd_insert

Called via TSX7, with index 1 pointing at a PDME <u>not</u> in the PD used list. Threads it into the PD used list, at the tail (most recently used) position.