SECTION IX

SERVICES OF PAGE CONTROL

This section describes the services that page control performs for the system. Foremost among these is the handling of page faults. Other services are performed for segment control, traffic control, and other supervisor subsystems. Although all of these services have been briefly described in Section V, the descriptions in this section explain the implementation of these services in terms of the mechanisms explained in Section VIII.

Other than the page fault handler, whose main path encompasses most of ALM page control, and the post purge function used by traffic control, all of these services are implemented in PL/I programs that operate on segments or portions of segments, calling the interfaces described in the previous section on each affected page, and multiplexing resultant wait events. The main memory and paging device reconfiguration services operate on portions of main memory or paging device instead of segments, again calling the ALM interfaces on each affected frame or record, and multiplexing the wait events.

All of the services of page control to segment control are implemented in the single PL/I program "pc", which, as noted in the previous section, has some code duplicating or subsuming functions of ALM page control were convenient.

PAGE FAULT HANDLING

The single most important function of page control is the handling and resolution of page faults. This code is implemented in the program page_fault, at the label "fault", transferred to by the fault vector directly, after the fault vector code has stored the SCU data for the page fault in pds$page_fault_data.

The essence of the page fault handler is to locate the page that must be paged in, and invoke the page-reading function to allocate a main memory frame and read it in. If successfully read in, the SCU data (machine conditions) is restarted; if I/O is started but not complete, the process must be made to wait for a completion of the I/O. If any of various error conditions prevail, the process must be caused to signal an appropriate condition, or restart the page fault to take a segment fault.

The most difficult task of the page fault handler is to locate the PTW for the page faulted on. Between the time that the processor actually takes the page fault and the page table lock is successfully locked to this process, it is possible that a "setfaults" operation (destruction of SDWs) might be performed on the segment containing the page, or the page of the descriptor segment containing the SDW for the segment might be paged out.

Although these events are highly unlikely, considering that the SDW must exist and be in main memory for the processor to observe that the PTW was faulted (modulo the associative memories), the page fault handler must be prepared to deal with these cases. The page fault handler needs the SDW for the segment to locate the page table for the segment and identify the particular PTW for the page on which the fault was taken, as only a segment number and computed address are supplied by the processor appending unit in the fault data.

The page fault handler depends upon non-local transfers by subroutines in the page-reading function; specifically, record quota overflow and out-of-physical-volume conditions in this function cause special action, including transfers back to the main path of the page fault handler.

The basic actions involved in handling a page fault are these:

1. Save all the processor machine conditions other than the SCU data, which was already saved. The page fault handler, unlike the segment fault handler, is the actual fault interceptor for this type of fault. Reset the processor mode register.

2. Mask to sys_level (it is not legal to accept interrupts during page control functions), and establish a stack frame on the base of the PRDS (processor data segment, wired per-processor stack) for the ALM page control environment (see "Stack Management," Section VIII).

3. Check for illegal conditions (page fault in ring 0 while the PRDS is in use as a stack) indicating system problems, crash if so.

4. Establish the ALM page control environment (initialize save stack, pointer register for the SST).

5. Try to lock the page table lock. Execute "Page Table Lock Waiting". Code if unsuccessful. (See "Page Table Lock Waiting," Section VIII.)

6. Perform the paging device housekeeping and replacement function, which ensures a small number of free PD records and currency of the PDMAP image on the bulk store (see Section VIII for details).

7. Determine whether this is a page fault taken on the descriptor segment when the processor needed on SDW (DSPTW APU cycle), a page fault taken pre-paging an EIS operand (PTW2 APU cycle), or a normal page fault on a page of a segment (PTW APU cycle). If none of the above, the processor is in error. In the first case, locate the page table for the descriptor segment of this process, and the page on which the fault was taken. The SDW for the descriptor segment is guaranteed to be in main memory. Proceed to step 9 in this case.

8. For a normal PTW or PTW2 cycle, try to obtain the SDW for the segment faulted on. If the descriptor segment is unpaged (as during initialization), there is no problem. Otherwise, check the PTW for the page of this process' descriptor segment containing the SDW for the segment on which the fault was taken. If this page is in main memory now, it can be read without taking a page fault. Pick up this SDW: a setfaults operation could have destroyed it at any time until this very instruction. If so, restart this page fault, abandoning the environment set up and unlocking the lock, causing the processor to take a segment fault by accessing that SDW. Otherwise, locate the page table and specific PTW for the page of the segment on which the fault was taken. (1)

9. Having located the PTW for the page on which the fault was taken, locate the AST entry for its segment.

(1) In the case of a PTW2 EIS prepage cycle, the computed address reported by the control unit in the SCU data must be adjusted one page up.

10. Check for two window situations involving some other process handling a page fault on that page before this process got the lock locked. In the first case the page is completely read in, and no page fault exists. In this case, unlock the lock, abandon the environment, and restart the machine conditions. The processor will then proceed to use that page. In the second case, the page could be being read in now, and is out-of-service on a read ("short page fault"). In this case, develop the wait event for the PTW and proceed to step 18.

11. Check for an error bit (ptw.er) set on by the interrupt side at the completion of a read from a previous page fault. If there was a read error, it turned on this bit (See "Error Strategy", Section VIII) and notified this process, so that it might take the fault over again and perform this step. Turn off the error bit, abandon the environment, unlock the lock, and transfer to the signaller (signaller$signaller) to cause the faulting process to signal page_fault_error.

12. Invoke the page-reading function (read_page) to find a main memory frame for the page and begin (and possibly complete) reading it in. This operation might possibly perform non-local exits in the case of record quota overflow (in which case that condition will be signalled in a manner identical to the signalling of page_fault_error in the previous step) or physical volume overflow (in which case the SDW will be faulted, the environment abandoned, the lock unlocked, and the machine conditions restarted to produce a segment fault). If the page-reading function encounters an RWS in progress on the page faulted on, set the abort bit ('pdme.abort) in the PDME for that page, causing the interrupt side to resolve the page fault (See "I/O Posting", Section VIII) with an "RWS abort", and proceed to step 18 to wait for this occurrence.

13. Meter this page fault. Compute the main memory usage charge of this process. Meter ring zero, directory, per-process, and level-one page faults. Compute the page-fault interarrival time histogram (displayed by print_paging_histogram) in the segment page_fault_histogram.

14. Execute the replacement algorithm write-behind function. This will cause writes to be queued, while the page read started by step 12 is in progress.

15. Now meter time spent processing this page fault.

16. If the page faulted on is not out-of-service, i.e., was either completely read in by step 12 or posted as complete by some actions occurring during step 14, the page fault is complete, and satisfied. Unlock the lock, abandon the environment, and restart the machine conditions. The process and processor will proceed to use that page.

17. The process must be made to wait for that page. If the page is involved in a bulk store transfer, "run" the bulk store (see "DIM Interface", Section VIII) until the page is no longer out-of-service, at which time go to step 16.

18. The process must be made to wait for a disk or RWS event. The page-reading function (or step 10) has developed the wait event. Transfer into the traffic controller environment as described in "Stack Management and Traffic Controller Interface" in Section VIII, causing the process to wait for this event, unlocking the lock, and abandoning the environment.

19. When such an event has occurred, or at least probably occurred, the traffic controller will transfer to page_fault$wait_return to restart the machine conditions. There is no page control environment or stack frame, and the page table lock is not locked. If indeed the interrupt side has posted this page, the process will resume and use the page. If indeed it has not (either the wakeup was spurious, as it is allowed to be, or the page has again been paged out in the window, the sequence of events starting at step 1 will be repeated.

SERVICES FOR SEGMENT CONTROL

Page control fills page tables and AST entries with information supplied by segment control, reports that information back to segment control as it changes dynamically, and performs operations upon those segments on behalf of segment control. The latter operations include truncating active segments, and evicting all of their pages from main memory and/or the paging device, so that segment control can deactivate the segments.

All of these functions, among others, are implemented in the PL/I program "pc". This program has available to it, via the transfer-vector "page", most of the functions in ALM page control described in the last section. Other than the activation-time service (fill_page_table), all of these operations are performed under the protection of the global page table lock. The program pc, as well as the other programs in call-side page control, use the entries pmut$lock_ptl and pmut$unlock_ptl to wire the current stack (3 pages of PDS), mask to sys_level, lock the page table lock, and undo all of these operations. In many cases, the entry page$cam is called before any unlocking (including that performed by the call-side wait coordinator) to make sure that any changes to PTWs are noticed by all of the system processors.

Activation-Time Service

(pc$fill_page_table)

This is the only fundamental service of page control that does not involve the page table lock. The entry pc$fill_page_table is called by segment control and the hierarchy salvager (among other parts of the system) to transform a file map in a VTOCE (see Section II) into a page table for use by page control. The routine is passed the AST entry pointer, the current length of the segment, and the PVT index to which the addresses in the file map refer. This routine does nothing more than translate the segment-control format addresses (see "Record Addresses" at the beginning of Section VI) and convert them into page-control format disk record addresses and null addresses, placing them in the PTW device address fields (mptw.devadd), initializing the rest of the PTWs as it goes. The PTW "first" bits, for the first-time PD performance optimization (See description of sst.ptw_first in Section VI) is initialized from that SST variable. A check is made, for each live address passed in, that it is indeed marked as "used" in the FSDCT (via a call to page$check_devadd). It is for this reason that the PVT index is passed as a parameter. This detects introduction of re-used device addresses into page control.

This service may be performed without the page table lock being locked. The caller guarantees that the segment whose page table is being filled is inaccessible, that no SDWs point at its page table, or will be made to point to it until after pc$fill_page_table returns. The check for reused addresses may also be made without the global lock locked; if the address is not reused, it will not be deposited in any possible window. If it is reused, it will stay that way whether or not the lock is locked.

File-map/Activation Attribute Reporting

(pc$get_file_map)

Segment control requires a reporting of the status of a segment and its addresses both at the time the segment is deactivated and at the time of the AST trickle. This information is used to update the VTOCE of the segment. The state of the addresses reported by this service to segment control is critical: it is a basic feature of the address management policy (see Section VII) that no nulled address ever be reported to a VTOCE file map. Thus a critical part of the file-map reporting service is the determination of whether or not an address should be reported to segment control at all. Part of the information returned to segment control is a list of nulled addresses that are to be deposited (returned to the free pool). The activation attributes of the segment are reported to the caller by filling in a complete copy of the AST entry for the segment, from a copy made under the protection of the page table lock. This copy must be made under the page table lock, in order for the "records used", "current length", and other fields to be consistent with themselves and with the list of addresses and list of addresses to be deposited that are returned.

Another action performed by the activation attribute reporting service is the maintenance of the "date-time-used" and "date-time-modified" fields in the AST entry. These fields are updated conditionally, depending upon the transparency attributes of the activations (see Section II), and the "file modified switch" (aste.fms).

All live addresses are reported to the VTOCE file map buffer passed in. Wholesale null addresses (representing no assignment of a record of disk) are also reported to this file map. The action taken for a nulled address depends upon several factors. A nulled address found in a PDME or CME (page on paging device or in main memory) must remain there; as long as a page has a frame of main memory or a PD record associated with it, it must have a disk record associated with it. A special null address (get_file_map_vt_null_addr, see "null_addresses.incl.pl1") is reported to the VTOCE for that page. The VTOCE will record no assignment for that page, as the nulledness of the address implied that the record of disk does not contain the page. A nulled address found in the PTW implies that the page has no main memory frame or paging device record associated with it (only disk addresses can be nulled). Normally, the action taken in this case is to report the address, not to the file map, but to a list of such addresses returned to the caller (the deposit list). The PTW is changed to contain a null address (get_file_map_vt_null_addr, again), and the caller is responsible for depositing all of the addresses in that list once it is known that the VTOCE has been successfully updated with the record address being deposited no longer in it. However, there is a class of circumstances in which the file-map reporting function may be inhibited from "culling" nulled addresses in this way. In these cases, nulled addresses in PTWs are left intact, and not reported to the deposit list of the caller. The caller may specify this behavior by passing the pointer to the deposit list as a null pointer. This action is also taken for segments with the switches aste.ehs and aste.dnzp both on. Such switches are set for hardcore segments in the normal AST used lists (and thus subject to AST trickle) which are prewithdrawn (such as the PDS of most processes). This action makes sure that prewithdrawn addresses stay withdrawn, i.e., are not deposited. See "Special Casing of Per-Process Hardcore Segments" in Section IV for motivation for this action.

The procedure pc$get_file_map is called with a pointer to the AST entry about which information is sought, a pointer to an ASTE image into which the AST information is to be copied, a pointer to a file map area in a VTOCE, into which the file map is to be placed, and a pointer to an array into which to put the deposit list. (As stated above, this pointer may be null). It returns, in addition to filling up the ASTE image, file map, and deposit list, a count of addresses put in the deposit list.

The procedure pc$get_file_map is also responsible for converting the page control format addresses into segment-control format (see Section VI), and turning off the bits aste.fms and aste.fmchanged (see Section II), indicating that any modification or file-map change for the segment has been noticed, and any further modification must be noticed independently.

Deactivation Service

(pc$cleanup)

At the time a segment is deactivated, any pages it may have in main memory or on the paging device must be evicted from these media. This must be done to satisfy the definition of a non-active segment, and to stabilize the state of the AST entry and file map.

The routine pc$cleanup is supplied a pointer to an AST entry for a segment to be so processed. The caller has ensured that no agency can bring pages of this segment in, either by having performed a "setfaults" operation on the segment, or being the only agency that has ever had access to the segment.

This routine is a prime example of routines that use ALM primitives and the multiplex wait protocol to process the pages of a segment in parallel, achieving state transitions by deterministic step.

With the page table lock locked, the following actions are performed for each page of the segment. The actions are repeated by reiterating over the segment until all pages of the segment are off the paging device, not undergoing RWS or paging I/O, and out of main memory. All addresses at that time will be in the PTWs.

1. Any page that is out-of-service (being read in or written out, perhaps by an earlier loop) has its wait event remembered, for potential waiting via the multiplex wait protocol.

2. Any page in main memory that is not out-of-service must be evicted; if it is modified (ptw.phm) the page-writing and purification function of ALM page control is invoked to purify it. If this puts it out-of-service, the PTW wait event is remembered for the multiplex wait protocol.

3. Any pure page is evicted by turning off its access bit and clearing the system caches and PTW associative memories via a call to page$cam_cache. If the page was modified in this window, restore the access and go back to step 2. The page then requires writing. If the page was successfully evicted, perform eviction cleanup (See Section VIII) not by a call to cleanup_page, but by inline PL/I code.

4. If the page still has a paging device record associated with it at this point (one may have actually been assigned in step 2, but this is rare), invoke the PD eviction subroutine (flush_one_pdrec) of pc to start an RWS, evict the page, or remember an RWS wait event as appropriate.

When all of the steps above have been performed for every page in the segment, and no steps (1, 2, or 4) remembered a wait event, the cleanup is complete.

Call-Side PD Eviction Subroutine

(flush_one_pdrec in PC)

This powerful subroutine is called by several services in the program pc, notably the deactivation service, PD reconfiguration service, truncation, post-crash-flush and shutdown services.

It is called with the variable "pdmep" pointing to a PDME describing a PD record that is to be vacated. The entries "flush_one_pdrec" and "delete_one_pdrec" differ only in the actions taken at the time the PD record is taken out of use; in one case the PDME is returned to the free list, and in the other case it is marked as deconfigured. Both of these entries evict the page from the paging device. The entry "truncate_one_pdrec", on the other hand, causes the destruction of the page, and the freeing of its PD record.

This subroutine, when not "truncating", starts an RWS for a PD record (via a call to page$pd_flush), which is modified with respect to disk, and not already undergoing RWS. It sets the multiplex wait variable "ind" to wait for any RWS that it starts (and does not find finished), or for any page out-of-service for paging I/O. For pages on the paging device not modified with respect to disk, it updates CMEs and PTWs, and frees or deconfigures the PDME.

One form of eviction from the paging device that is unique to this subroutine is that performed for pages in main memory (although not undergoing paging I/O). The paging device replacement algorithm does not evict pages from the paging device which have copies in main memory, because this is considered evidence of recent use. When eviction for such pages must be performed, however (as is the case in all call-side entries that need it), it is very simple to effect. The disk address from the PDME simply replaces the paging device address in the CME. The PDME "modified" bit (pdme.mod) is "or'ed" into the PTW "modified" bit (ptw.phm), using a key-line instruction. This causes the page to be written out to disk when it is evicted from main memory, in the case where the paging device contents were different from the copy of the page on disk, but the same as those in main memory.

This subroutine must not free PDMEs for PD records that have undergone RWS on behalf of the post-crash PD flush.

This subroutine sets "notify requested" bits in CMEs and PDMEs when it returns a wait event. This is superfluous, as the call-side wait coordinator will do this if such wait event is actually passed to it.

Truncation Service

The truncation of segments is performed for both segment control (from the Segment Control Truncation function (See Section IV)) and for supervisor subsystems that deal with non-hierarchy segments, in order to free their disk record resources.

Truncating a segment to length n (n given in pages) involves truncating all pages of page number equal to or greater than n. Truncating a page means associating zeros with the contents of that page; in fact, the actions performed to truncate a page in main memory are identical to those taken by the page-writing function (see Section VIII) when a page of zeros is discovered. Truncating a page which is neither on the paging device nor in main memory consists of no more than nulling its disk address (and updating the necessary ASTE quantities and quota cells). Recall that a nulled address is paged in by the page-reading functions as a page of zeros. Truncating a page on the paging device, whether or not it is in main memory, involves freeing the associated PD record by placing the PDME for it in the PD used list.

Truncating a segment consists of little more than performing the above actions, as is the case for each page. For pages that have paging I/O going on (ptw.os is on), the completion of this I/O is awaited via the multiplex wait protocol, For pages on the paging device, the call-side PD eviction subroutine (see preceding) is invoked (at the entry "truncate_one_pdrec"). Among the actions taken by this subroutine is the remembering of any RWS in progress on that page, for later waiting via the multiplex wait protocol. When all pages in the segment have been processed and no wait events remembered, the truncation is complete.

A large part of the complexity of the segment truncation primitive is the determination of whether or not a page being truncated was charged against quota. Basically, any page in main memory is charged against quota. Any page with a live disk address is charged against quota. Those pages with null addresses, or with nulled addresses but not in main memory, are not charged against quota.

At the end of the processing, the ASTE fields describing the number of records used and the current-length of the segment are updated. If quota checking is not inhibited for this segment (i.e., aste.nqsw is off), the quota utility quotaw$cu_for_pc is called to adjust the quota account against which the segment's record quota is charged. If any pages were actually truncated, the file-map-changed bit (aste.fmchanged) is set, indicating that segment control should update the VTOCE as soon as convenient, for addresses can be deposited, and must be removed from the VTOCE. Segment control's VTOCE update function will do both these things.

The page control truncation service does not require that no other agency in the system be creating pages while it is trying to truncate them. That is the problem of the subsystem or user code which invoked the storage system's truncation facility, not of page control. The only issue here is that the truncation service must be quite careful to multiply count pages it truncates multiple times. It is impossible for malice or accident to force the truncation service into a loop by so doing: only when the page table lock is unlocked while pc$truncate waits can such pages be created. The user cannot force paging-ins of nonzero pages in the truncated region, or their paging-out, or RWSs which are the only activities that will cause pc$truncate to wait.

The arguments to pc$truncate are the AST entry pointer of the segment and the length (n) to which it is to be truncated.

Boundsfault Service

(pc$move_page_table)

The segment control processing of a boundsfault usually involves the allocation of a new ASTE/page table pair for a segment, and the establishment of that ASTE and page table as the sole ASTE/page table for that segment. From the segment control side, the most critical operation here is the hashing of the old one out of the AST hash table, and the hashing of the new one in.

However, when such an operation is performed, if there are pages of the segment on which the boundsfault has taken place in main memory or on the paging device, there are page control data bases that describe the original ASTE and page table (PTWs) of the segment. Sometime during the processing of a boundsfault, page control must be invoked to construct a valid page table for the new ASTE, and modify all page control data bases that referenced the old ASTE/page table to reference the new one. This service is provided by pc$move_page_table, called with the two AST entry pointers involved.

It is also critical that all of the page control maintained data items in the ASTE be copied from old to new during the same locking of the page table lock (only one such locking will be required, there is no I/O involved) as that which the page table is reconstructed. This must be so in order that the interrupt side will reference the correct ASTE should any I/O on this segment complete, and so that any paging-out activity will do the same. The caller of pc$move_page_table (the boundsfault handler of segment control) ensures that neither ASTE is accessible, i e., no process can access the segment on which the boundsfault has taken place.

The task of the page control boundsfault service is simple: all pages in main memory or on the paging device or on disk remain exactly as they are, whether or not I/O or RWSs are in progress on them. The essence of the task is to walk the old page table and new page table in parallel, chasing down any CMEs or PDMEs designated by the PTWs in the old page table, and changing the pointers in these CMEs and PDMEs to point to the new page table. The old PTW contents are copied to the new PTWs. The ASTE relative-pointers in the CMEs are similarly modified. The PTWs in the extent of the new page table beyond the extent of the old are similarly modified. The page table lock remains locked during the entire operation, ensuring that no process can use the data objects or change their state while they are being modified. Before the lock is unlocked, the entire ASTE contents (other than its threads and pool number) are copied from the old to the new ASTE.

One subtlety of the boundsfault service requires some clarification. The relative offset of PTWs into the SST segment is used as a wait event ID by processes awaiting the completion of non-RWS paging I/O (See "Wait Events", in Section VIII). When the page table has successfully been moved, the interrupt side will post any I/O which completes after that point by notifying the event ID associated with a new PTW. Thus, processes waiting for the page which began this waiting before the page table was moved are no longer waiting for the correct event, and will not be notified. Thus, the boundsfault service explicitly notifies any PTW event for which the CME notify-requested bit is on. This causes any process waiting for a PTW event associated with the old page table to run; when it successfully locks the page table lock, it will retry whatever it was doing, either via taking a segment fault or a page fault, and ultimately find the new PTW, and go to wait on that.

Modified-Switch Setting

(pc$updates)

Directories are normally activated with the transparent-modification attribute (see Section II for more illumination). This means that changes to the contents of the directory do not cause the file-modified switch of the directory to be set. This, in turn, means that the date-time-modified of a directory or its superiors is not advanced solely by modifying a directory. Although this convention dates from times when date-time-used was stored in a directory (it is now in the VTOCE for a segment) and change to this field had to be made without updating the date-time-modified of the directory, there is still a small class of operations (segment moving and online-salvaging) which modify directories in ways such that the directory date-time-modified should not be advanced.

The date-time-modified of a directory is defined recursively as the latest date-time-modified of any segment or directory under it, or such time that the directory was explicitly modified by directory control. In the case when segments are modified, the page control page-writing function notices the "modified" bit in the PTW (See "page-writing function" in Section VIII), and turns on the file-modified switch (aste.fms) in the ASTE of that segment and all of its superior directories (this bit is reported by the file-map and activation-attribute reporting service described earlier in this section). For the case of explicit modification of directories by directory control, an address-space management utility (sum$dirmod), to update certain fields of the directory. One of the actions taken by this program is to obtain an AST entry pointer for the modified directory via a call to "activate" (See "Significance of Activate," Section IV) and pass it to pc$updates. This entry, with the page table lock locked, does no more than chase up the ASTEs from that ASTE on up setting ASTE "fms" (file modified switch) bits explicitly.

POST-CRASH PD FLUSH

The management of the paging device is such that it contains information (copies of pages) that is not identical to copies of the same pages on disk. Records containing such pages are called "modified" PD records. In order to evict such pages from the paging device, a read-write sequence (RWS) must be performed. Part of the task of shutdown, normal or emergency, is to flush the paging device, i.e., evict all pages from it. This implies read-write sequences for all "modified" PD records. However, should a successful shutdown not be possible, the "modified" records of the paging device contain information duplicated nowhere else. The next bootload of Multics must copy the contents of these records back to the disk records to which they belong. This is known as repatriation of these pages. Repatriation of pages that had nulled disk addresses also involves resurrection of these addresses, implying modification of VTOCEs. The post-crash PD flush is the page control service that performs these tasks.

The paging device may be said to come in instances. An instance of a paging device is the paging device, its map, and all the pages which have ever been on it from the time that map is initialized, to the earliest of a successful shutdown, or flushing of the last record off of the paging device by post-crash flush, or abandonment by the operator "force_pd_abandon" ring-1 command (see the Multics Operators' Handbook, Order No. AM81). An instance of the paging device exists during only one bootload if that bootload shuts down successfully. Otherwise, it may exist during two or more bootloads. An instance of the paging device is uniquely identified by the "paging device time," the field pdmap_header.time_of_bootload, set to the clock time at which the paging device map was initialized. This field, along with the rest of the paging device map, is written out to the first few records of the bulk store every second by the PD housekeeping function in the page-fault handler. It is also written out by explicit calls to pc$write_pdmap.

An instance of a paging device that was created during the current bootload is said to be an active paging device; the system is said to have an active paging device. The bit fsdct.pd_active in the FSDCT header indicates this. An instance of a paging device that was created during some bootload other than the current bootload, and not successfully flushed (i.e., successful shutdown was not achieved) is called an unflushed paging device. When a hierarchy (or a bootload) has a paging device in this state, the system is said to have an unflushed paging device. The bit fsdct.pd_unflushed is on when this is the case.

Whenever a physical volume is accepted by a system with an active paging device, or an instance of a paging device is created (the paging device becomes active) during a bootload, the physical volume is said to have been exposed to that instance of the active paging device. Whenever a physical volume is accepted, the paging device time of the instance of the active paging device, if there is one, is written to the label of that volume before any segments on it are allowed to be activated. Whenever a paging device is made active during a bootload, a call is made (to the program fsout_vol, for each volume, see Section XIV) to write the paging device time to the labels of all volumes before that paging device is actually made available to the PD record allocator. Thus, any physical volume contains in its label the PD time of the last instance of the paging device to which it was exposed.

The label of the root physical volume (RPV) contains a bit (label.pd_active) which says whether it, and therefore the entire hierarchy which it commands, was exposed to an active paging device, this bit being cleared when the system is successfully shut down. If the system comes up after a crash and this bit is on, then the system must have an unflushed paging device, otherwise the system would have been successfully shut down and that bit cleared. Thus, the paging device map is read from the bulk store, and the PD time in the PDMAP header compared to that of the instance of the paging device to which the RPV was last exposed. If these are not the same, the paging device contents have been damaged (probably by the use of another hierarchy) since that RPV was last used (and not shut down). The system will not come up in this case; the operator must zero the paging device. If the system finds the paging device time on the bulk store zero, when the RPV was indeed exposed to an active paging device and not shutdown, it implies that the operator cleared it. A message is typed, and a new instance of the paging device is created. If the times indeed match, however, the system has an unflushed paging device, which must be flushed. The bit fsdct.pd_unflushed is turned on to this effect. All of the records of the paging device that are not "modified" have their PDMEs cleared (set to zero). Those marked as "deleted" by the PAGE CONFIG card (see "PD Reconfiguration" later in this section) are deleted. The state of the "modified" PDMEs left by these actions is regularized. There is no PD used list on an unflushed paging device. The map is written out in its new state. The records on the paging device will be repatriated as volumes are accepted. The manipulations described above are all performed in the program "init_pvt".

Whenever a physical volume is accepted by the system, it can tell whether or not it has been successfully demounted. Shutdown, it will be recalled, demounts all volumes (See Section XIV). Whenever a physical volume that has not been shut down is accepted, the physical volume salvager is invoked by volume management to salvage it. This physical volume salvaging, among other things, reconstructs the map of free addresses, and checks each VTOC entry (VTOCE) for consistency.

Whenever a physical volume that has not been successfully demounted is accepted by a system with an unflushed paging device, there exists the possibility that that volume was exposed to that instance of the paging device. If the volume was shut down successfully, it cannot have any records on any instance of the paging device. Only volumes present at the time of the crash can have records on this instance, the unflushed, current instance, of the paging device. Those volumes are exactly the set of volumes not successfully demounted which were exposed to this instance of the paging device. Whenever a volume that was not successfully demounted is accepted by a system with an unflushed paging device, comparing the PD time in the label of that volume to the PD time of the current, unflushed, paging device tells whether or not this is the case. Such a volume is said to have been exposed to an unflushed PD.

Whenever a volume exposed to an unflushed PD is being salvaged, records on that paging device will be repatriated to that volume. The task of identifying these records is facilitated by the recording of the physical volume table index of the volume containing the page in the PDMAP entry. This PVT index identifies the drive on which the volume was mounted during the bootload that crashed, which may not be the same as during the current bootload. However, the physical volume table index, as well as the PD time, is recorded in the pack label at the time the volume is accepted. Since the volume is guaranteed not to have been successfully demounted, it is impossible that any other volume could have had that PVT index after that during that bootload, and hence have pages on the unflushed paging device. Thus, by comparing the PVT index in the volume with that of each disk record stored as the "devadd" in a PDMAP entry, it can be determined precisely whether that PDME describes a record to be repatriated to this pack, and if so, to what disk address.

Repatriating the pages is only half of the task. Many of the "modified" pages on the paging device contain pages that were never written to disk; their entries in the file map of their segments' VTOCEs contain null addresses. Thus, simply writing back these pages to the disk is not enough, as a fault on that page will produce zeros, as the address in the VTOCE is null. Thus, in effect, such repatriations are resurrections; live addresses must be reported to the VTOCE. It is for this purpose that segment unique ID and page number are stored in the PDMAP entry. As each VTOCE (each segment) of a physical volume exposed to an unflushed paging device is processed by the physical volume salvager, a special service of page control (pc$flush_seg_old_pd) is called, passing the address of the file-map region of the VTOCE image, the old and current PVT indices of the physical volume, and the UID of the segment whose VTOCE is being processed as input parameters.

The entry pc$flush_seg_old_pd scans the entry PDMAP looking for PD records containing pages belonging to this segment; such entries have a matching UID, placed there by the PD record allocator (see Section VIII). For each such entry, an RWS is initiated by the use of the call-side PD eviction subroutine, flush_one_pdrec (see earlier description in this section). All of these RWSs are performed in parallel, and waited for in parallel via the multiplex wait protocol. A special bit (pdme.flushing) is turned on before each RWS is initiated, so that the interrupt side will neither clear nor free the PDME (see "I/O Posting", Section VIII). In order for the RWS mechanism to work, the PVT index in the PDME must be the PVT index of the volume in the current bootload, if the volume has moved. Thus, before invoking the call-side PD eviction subroutine, pc$flush_seg_old_pd saves the old PVT index in mpdme.save_old_pvtx, and places the new one in. This is done so that should the system crash during the processing of this segment, the next bootload can detect this (pdme.flushing will be on), and cleverly place the old PVT index back.

As the interrupt side completes each such RWS, it leaves the PDME intact. In all cases except the case of RWS (read or write) error, the disk address in the PDME will be a live disk address (RWS completion always causes a resurrection, if the disk address was nulled). However, a null address will have been left by the interrupt side if there was an error. In all cases except the last (RWS error), the live address from the PDME is moved to the appropriate slot in the VTOCE file map passed in as an argument (pdme.pageno says which slot), and two output parameters, representing segment current length and "records used", are adjusted if a resurrection took place. The PDME is then zeroed, but not freed.

The post-crash PD flush repatriation procedure substantially increases the time required to do a physical volume salvage, as the whole PD map must be scanned for each VTOCE processed.

The placing of a UID in a PDME ensures that there are no windows between the last time the map was written out and the last time data was written to that PD record. Were this not the case, the wrong data might be flushed to some segment.

After all VTOCEs have been processed by the physical volume salvager, a special primitive (pc$cleanout_old_pd_pv) is called to clear PDMAP entries for "parasite" segments (i.e., those with no VTOCEs), such as descriptor segments on the RPV (see Section VII). This primitive also checks that no records exist on the paging device which belonged to the volume being salvaged; they should all have been repatriated. If there are any, very little can be said or done about them, and nothing would be gained by crashing the system. An informative message about the curiosity is typed out.

SHUTDOWN AND DEMOUNTING SERVICES

The aims of both shutdown and demounting are to ensure that the paging device and main memory contain no pages of a set of physical volumes; in the case of demount, it is one physical volume. In the case of shutdown, it is all of the volumes present. Demounting causes this to occur by deactivating all of the segments on the volume. Shutdown, however, although it goes through the demount procedures for all volumes present, does not attempt these deactivations.

Shutdown flushes the paging device (evicts all pages on it) as early as possible. This is so that the system should not have an unflushed paging device, should shutdown fail. Obviously, only active paging devices can be flushed. The entry pc$pd_flush_all exists for this purpose. It calls the call-side PD eviction subroutine flush_one_pdrec (see earlier description in this section) to flush each page off of the paging device (initiating RWSs if modified), and uses the multiplex wait protocol to multiplex the wait events. When this routine returns, the paging device may be declared inactive.

Shutdown also flushes all of main memory before doing its update_vtoce loop; this is so that any disk record addresses for pages in main memory (the paging device has been flushed, as above, at this time) are resurrected prior to the VTOCE updates. The routine pc$flush is called for this purpose. It calls the page writing/purification function in ALM page control (See description in Section VIII) to initiate writes on all pages that are modified with respect to main memory. All I/Os are awaited (whether or not this action started them) via the multiplex wait protocol. This action also causes all pages of zeros in main memory to be evicted, nulling their addresses.

Normal and emergency shutdown call the primitive pc$write_pdmap several times to write out the PD map to the bulk store when significant changes to its state are made. Writes performed by this primitive are done via using the segment pdmap_seg as an abs-seg over the bulk store; such writes are done via calls on the page-writing function in ALM page control, and are posted normally via the interrupt side. This does not conflict with writes to the map performed by the PD housekeeping function, which may even be going on simultaneously; these writes do not involve PTWs or CMEs, and will not even be reported by the bulk store DIM to the interrupt side upon completion. The function is also used by PD Reconfiguration, see Section X.

Volume demounting does not require any special services from page control; all of the flushing of pages out of main memory and off of the paging device are performed by pc$cleanup, invoked by segment control when segments on that volume are deactivated (see Section IV). However, a special entry in page control is called by the volume demounting function after all segments have supposedly been deactivated. This entry, pc$check_pd_demount, does no more than check that no pages belonging to that volume are still on the paging device. This is solely as a check for bugs; it should never be the case that there are such records.

RECORD ADDRESS DEPOSITING SERVICES

pc$deposit_list
pc$list_deposited_add
pc$truncate_deposit_all

Page control, as the maintainer of the FSDCT bit maps for mounted volumes, is charged with the depositing of addresses on behalf of segment control and other agencies.

The entry pc$deposit_list is called with a "deposit list", an array of addresses to be deposited, and a PVT index identifying their volume. Such a "deposit list" is produced by the file-map reporting service (pc$get_file_map), and by the segment control segment truncation facility in the program truncate_vtoce. The number of entries in this array is also a parameter. Basically, this entry does nothing more than iterate over the array supplied and call the withdraw/deposit mechanism in the program free store (See "Individual Mechanisms" in Section VIII) with each address and the PVT index. This operation is performed without the protection of the page table lock: depositing is a unitary operation that involves no races, as only one process can deposit a given address at one time.

The entry pc$list_deposited_add is an entry that performs that function of the file-map reporting service which is the reporting of nulled addresses and their replacement in PTWs by null addresses. This entry places the addresses so gathered into a "deposit list", such as that accepted by pc$list_deposit above. This operation must be performed under the protection of the page table lock. The criteria for reporting an address are the same as those in pc$get_file_map, i.e., the address must be nulled and in a PTW. Addresses so reported are replaced in the PTW by the null address "list_deposit_null_addr" (See null_addresses.incl.pl1). This entry is used by code dealing with non-hierarchy segments, such as some initialization code, and by the segment mover (See "Segment Moving" in Section IV).

The entry pc$truncate_deposit_all is a macro operation consisting of successive calls to pc$truncate (to zero length), pc$list_deposited_add, and pc$deposit_list. It is used to destroy RPV parasite segments (e.g., PRDSs and descriptor segments). It is supplied an AST entry pointer as an argument. There is no window between the truncation and the depositing: these segments have no VTOCEs, and are not under consideration by any subsequent bootload.

PAGING DEVICE RECORD DELETION

The paging device reconfiguration service is described in Section X, "Peripheral Services of Page Control", as it does not interact with the mainstream of page control, and in general only deals with deconfigured paging device records. However, one part of the paging device reconfiguration software involves taking paging device records out of use. This involves the use of the services of the kernel of page control.

All paging device reconfiguration is managed by the program delete_pd_records, which wires itself via the wire_proc mechanism (See Section X) when invoked. In all cases except the use of the "delpage" operator command, it deals with unflushed paging devices and deconfigured records. It does not involve PTWs, CMEs, or pages of segments.

However, the "delpage" operator command may involve the eviction of pages from in-use paging device records. In this case, delete_pd_records, which is at that point running masked and wired with the page-table lock locked, invokes the entry pc$delete_pd_records with the first index and number of paging device records to be deleted. Using the call-side PD eviction subroutine, flush_one_pdrec (called at the delete_one_pdrec entry) (See earlier description of this subroutine), this routine evicts pages, starting RWSs where necessary, and waiting for all paging and RWS I/O via the multiplex wait protocol.

The eviction of pages from PD records performed for record deletion is different from those evictions performed for deactivation, truncation, or the PD replacement algorithm, etc., insofar as the PDME for the PD record is not to be threaded into the PD used list, but left out, with a word of all ones set in its thread word. This marks the "deconfigured" state of the paging device record. When a nonmodified page is evicted by delete_one_pdrec, this subroutine performs this deletion. Otherwise, the bit pdme.removing is set on before or during RWS, so that the interrupt side (See "I/O posting" in Section VIII) will deconfigure the record instead of threading the PDME into the used list.

Paging device records are also deleted automatically by the interrupt side when paging device I/O errors have occurred; see "Error Strategy" in Section VIII.

FORCED SEGMENT I/O AND WIRING

(pc_wired)

Several agencies in the system have the need to "wire" portions of segments (make their pages nonremovable from main memory). In some cases, this is accomplished by turning on "wired" bits in the PTWs for the affected pages and simply touching them. This technique is generally used to wire regions of stacks. A less ad-hoc facility is available though, through the entries pc_wired$wire, pc_wired$wire_wait, and pc_wired$unwire. Typical uses of this facility are for wiring data bases to be used at interrupt time by facilities that are not always enabled (such as the ARPANET software), and by wire_proc, the manager of procedure-wiring requests (described in Section X), which temporarily wires procedures that must not take page faults.

The program pc_wired implements all of these functions, along with a few others described below. In all cases, it is passed the AST entry pointer for some segment (it is the caller's responsibility to ensure that the segment is either a supervisor segment or cannot be deactivated while pc_wired is operating upon it), a first page number, and a number of pages to be read/written/wired/unwired. In all cases, a number of pages of -1 indicates that all pages from the "first page" specified to the end of the segment are to be read/written/wired/unwired.

The service provided by pc_wired$wire_wait is the most often used. It wires the pages of the segment specified, if they are not already wired, and does not return until they are all in main memory. It operates by turning on all of the "wired" bits in the affected PTWs that are not already on, and initiates reads via calls to the page-reading function (See Section VIII) via the transfer-vector page$pread. All I/O operations on these pages, whether noticed or started by this module or reported back by page$pread (which may include FSDCT pagings, RWS events, etc.) are awaited via the multiplex wait protocol, until all specified pages are in main memory, with no I/O in progress on them.

The service provided by pc_wired$wire is similar, but it does not retry calls to ALM page control or wait for I/O completions. Thus, its effect is little more than to turn on all of the wired bits involved and start some of the I/Os. This service is not particularly useful, and is not used.

The service provided by pc_wired$unwire is commonly used. It simply unwires the pages wired by either of the two above entries. In all cases, this is nothing more than turning off the PTW "wired" bits.

The module pc_wired also provides a set of services to perform paging I/O on demand upon segments. These are used by the physical volume salvager to pre-page (i.e., start asynchronous paging-in) segments used to address VTOCs, and by the segment mover to force zero pages to be noticed (and thus have their addresses nulled) by the page-writing function (See Section VIII). A special form of this service is available to the directory control directory unlocking primitive, via pc_wired$write_wait_uid. This entry is used when a directory is unlocked which directory control knows to have been modified; it causes all the modified pages to be written from main memory (perhaps to the paging device) as a hedge against crashing. It is different from the service provided by pc_wired$write_wait in that the caller makes no guarantee that the AST entry pointer provided will remain valid during the operation of pc_wired; therefore, a segment unique ID (UID) is supplied by the caller so that pc_wired can check the AST entry each time the page table lock is relocked, to ensure that it still designates the same segment.

The entry pc_wired$write_wait is the most generally used. Given that the caller ensures that no process may be modifying the segment, it ensures that no modified pages of the segment (in the range specified) exist in main memory when it returns. The entry pc_wired$write performs similar actions, but does not wait, and makes no statement about the state of the segment when it returns, and thus is not used.

The entry pc_wired$read is used to start page-reads for all pages in the range specified. It makes no guarantees about when these reads will be complete; this is used solely as a performance optimization feature, for those supervisor subsystems that can anticipate their page reference patterns. There is no entry pc_wired$read_wait; if there were, it could not possibly guarantee that pages which it had read would stay in main memory when it returned, for any paging activity at all could evict them. The concept of reading pages in nonevictably is the concept of wiring, treated above.

The entries pc_wired$read, pc_wired$write, and pc_wired$write_wait all operate by calling the page-reading and page-writing functions in ALM page control, and iterating in the "wait" cases via the multiplex wait protocol.

ABS-WIRING SERVICE

(pc_contig)

Peripheral device operation via the IOM (Input-Output Multiplexer) requires contiguous regions of main memory for data buffers. The IOM provides a facility whereby arbitrary user-supplied channel programs may be run in a given region of main memory, preventing them from damaging other regions of main memory via a per-channel limit register. The same facility also relocates addresses appearing in such channel programs with respect to the base of the region, such that the writer of such a channel program need not even know where in main memory the channel program (and the data) will appear. The use of this facility is managed by the I/O interfacer.

A critical part of this facility is the ability to acquire successive page frames of main memory that can be made to form a contiguous region. When storage system segments are to be used as buffers for the IOM, they must be paged into such regions of pages, in address order, and not be evicted from those page frames for any reason, including deconfiguration of memory. Such pages may not be moved around main memory, as is done by some of the functions described in Section VIII. Such pages are said to be "abs-wired", as are the segments to which they belong at the time that their pages are in this state.

The use of abs-wired buffers, for the IOM (and the FNP6600 Communications Processor bootload software, through the IOM) is managed by a program called the I/O buffer manager (iobm). This program calls the page control segment abs-wiring service to allocate regions of main memory and abs-wire segments into them. It uses timers and request queues to schedule re-use and unwiring of these buffers. The I/O buffer manager also performs the unwiring of these buffers when that act is appropriate; it turns off CME abs-wired bits and PTW wired bits, operations that need not be protected by the page-table lock.

The program pc_contig is responsible for abs-wiring portions of segments. To abs-wire a portion of a segment, a number of usable main memory frames equal to the number of pages of the segment to be abs-wired must be found. A main memory frame is usable if it is in a non-deconfigurable system controller, is not deconfigured already, is not already in use by an abs-wired segment, and is in the first 256K of main memory. All frames found must be in the same system controller. (The issue of the first 256K involves an IOM design issue known as "backup list service".

The entry pc_contig$wire is called with the AST entry pointer of the segment whose pages are to be abs-wired, the number of the first such page, and the number of such pages. It returns a core map entry pointer to the first core map entry of the region into which it allocated and paged in and abs-wired the pages of the segment, from which the I/O buffer manager computes the main memory address of the region. This pointer is returned as null if the requested allocation could not be performed. A flag is also passed indicating whether or not this entry has been called on the interrupt side; currently, it never is.

The entry pc_contig$wire locks the page table lock and scans the core map for a sufficient number of usable main memory frames; if there are not enough, it tries several times to call the I/O buffer manager to release any frames it possibly can which it is holding. Only if this repeatedly fails is the caller informed that the requested allocation cannot be performed.

Once a region of main memory is decided upon, all pages currently residing there are evicted via the demand eviction function described in Section VIII. All I/O and RWSs in these frames are waited out. The pages of the segment to be abs-wired are read in via the page abs-wiring function described in Section VIII. For each page, this reading does not commence until the previous contents of the frame have been evicted, and I/Os already in progress there waited out. All of these operations are paralleled and waited for in parallel via the multiplex wait protocol.

Another interface to the abs-wiring service is maintained for historical reasons in the program pc_abs, at the entries pc_abs$wire_abs and pc_abs$unwire_abs. These entries are called with AST entry pointers, number of pages to be wired or unwired, and the number of the first such page. In the wiring case, this entry does nothing more than call pc_contig$wire. In the unwire case, the CME "abs-wired" bits and the PTW "wired" bits are turned off, again not requiring the protection of the page-table lock. This set of interfaces is currently used only by the ARPANET software to create buffers for the Interface Message Processor (IMP).

MAIN MEMORY DECONFIGURATION SERVICE

The Multics Dynamic Reconfiguration Software (See the Multics Reconfiguration PLM, Order No. AN71) provides the ability to take single frames of main memory out of use, and to take many out of use in order to take an entire system controller out of use. The commands which perform these activities are the "delmain" and "delmem" commands (see the Multics Operators' Handbook, Order No. AM81). Taking frames out of use in this way is performed by the page control main memory deconfiguration service provided by the entry pc_abs$remove_core. The entire power of this program is derived from the demand eviction function described in Section VIII and the multiplex wait protocol.

The program pc_abs, when invoked at this entry, with the first frame number and number of frames to be deleted, starts off by making legitimacy checks; the system must be left with enough main memory to function, and no frame that contains an abs-wired page may be deleted. The program wires itself via the procedure temp-wiring service described in Section X, and locks the page table lock, masking and wiring its stack via pmut$lock_ptl (see Section VIII, "Lock Conventions").

The program iterates over the region to be deleted, assured of the legitimacy of the request, turning on the "removing" bit (cme.removing) in the core map entry for each main memory frame to be deleted. This ensures that the main memory frame allocator (find_core) will never allow this frame to be allocated to a page; this ensures the deterministic success of the eviction that follows. The demand page eviction function is invoked on each frame that is not already deleted, until all frames are deleted. The wait events returned by page$evict are multiplexed by the multiplex wait protocol. Each frame from which a page has been evicted, with no wait event, is threaded out of the main memory used list, and given a thread word of -1.

The program returns, unlocking the page table lock, unwiring its stack, and unwiring itself.

SERVICES FOR TRAFFIC CONTROL

Traffic control performs many services for page control, notably implementing the wait/notify mechanism by which the waiting for of many page control events occurs. Page control also performs three services for traffic control: the loading and unloading of processes, and the post-purging of a process.

Process Loading

(wired_plm$load)

The two critical pages of a process (the first page of the descriptor segment and the process data segment (PDS)) must be wired before a processor is allowed to run in that process. A process in this state is known as loaded. The loading of a process is performed at the time it acquires eligibility. The loading of processes is performed by the program wired_plm, which has as its sole entry point the entry wired_plm$load.

The process-loading function is different from any other service in page control insofar as it performs its task on behalf of some other process than the one in which it is invoked. The process-loading function is invoked from the traffic controller's "getwork" routine (with the traffic control lock unlocked) at the time a process is being made eligible. Since loading a process may involve waiting for the reading (or RWS completion) of the two critical pages, waiting must be performed if this is the case. The process that is currently running in the traffic controller cannot and should not be made to wait for these events, involved in the loading of some arbitrary process. Thus, the process that is being loaded is made to wait for the events involved in its own loading itself. The traffic controller will not try to run any process that is waiting for an event, whether or not that process is loaded. When an event is notified, the traffic controller will usually try to run a process that had been waiting for that event. However, if that process is not loaded, a call will be made to the page control process-loading function to achieve or continue the loading of that process. Only when the loading function returns the result that the loading of the process is complete does the traffic controller mark it as "loaded", begin to run it, and interpret notifications of wait events in the normal way.

Thus, when a process is made eligible (it is never loaded at the time it is made eligible) a call is made to the loading function to start as many operations as can be started in parallel to accomplish its loading. If the process-loading function returns the fact that the process is loaded upon return from this function, then that is the case. Otherwise, the process-loading function returns a wait event of some operation that it started which was not completed. Since the traffic controller will cause the process being loaded to wait for that event, and call the loading function back when that process is notified, the effect is that of the process-loading function being called back when that event has been notified. Thus, in effect, the process loading function is called in a loop for each loading, returning either a wait event or an indication of loading having been successfully achieved each time. It is called again and again each time it returns a wait event, after that event has occurred, until it returns an indication of complete loading. This is a strategy very much parallel to the simple and multiplex wait protocols used elsewhere in page control.

The loading function is invoked without the traffic controller lock locked. It locks the page table lock, and unlocks it when done. Since the traffic controller locks its lock, upon return from the process-loading function after the latter has unlocked the page table lock, there is a window between these two events during which the event handed back by the process-loading function might occur. Thus, the traffic controller "validates" such events by actually checking PTWs and PDMEs designated by such events for valid out-of-service or RWS states (See Section VIII for further discussion of this anomaly). If such an event is found to be "invalid", the process being loaded is set up so that the process-loading function will be called again for it as soon as possible, i.e., the process-loading function will be retried without any wait.

The code of the process-loading function itself is very simple: it develops wait event IDs for either of the two critical pages it finds out-of-service or undergoing RWS, and invokes the page-reading function of ALM page control (page$pread) to read in either of the two pages not in main memory, remembering the event ID of any event detected by this primitive. It returns to the traffic controller any of the wait events encountered in either of these ways; if there are none (both pages are in main memory), it returns to the traffic controller the fact that the process is loaded. This code also turns on the "wired" bits in the PTWs of the two critical pages if they are not already on; this is part of its contract, and ensures completion of the read operations in a deterministic number of steps.

Process Unloading

Process unloading consists solely of turning off the "wired" bits in the PTWs for a process' two critical pages. This operation, which need not be performed under the page table lock, is done by the routine "unload_old_process" in page fault, invoked solely by the traffic controller and returning to it.

Post-Purging

The post-purging service of page control is used as a performance optimizing algorithm to bias the page replacement algorithm in favor of replacing pages of a process that loses eligibility. This service is invoked by the traffic controller at the time that a process loses eligibility, for any process whose work class indicates that post-purging is to be performed. Part of the post-purging service also consists of estimating the "working set" of the process, used by the traffic controller in the decision to grant eligibility.

The basic task of the post-purging function is to scan the per-process trace list of pagings performed by a process (see Section VIII for the "per process page trace list") (this function runs in the process it is processing), and to classify the pages involved in the various paging-ins as part of the process' working set or not, and bias the page replacement algorithm in favor of their replacement in various ways.

The post-purge function is implemented in the ALM program post_purge, called by the traffic controller with the traffic control lock not locked. It locks the page table lock at the start of its processing, and unlocks it only at the end of its processing, before returning to the traffic controller.

The post-purge function considers each page reading in the trace list, between the last time the process was post-purged and the current time. It also makes an entry in the trace list, a "scheduling" entry, for use by the "page_trace" command. It considers six attributes of each page in the trace list, and performs up to four potential actions for each page based upon them. These attributes are:

1. The page being in main memory at the time it is seen.

2. The page being on the paging device.

3. The page being part of a per-process (aste.per_process on. segment.

4. The page having its "used bit" (ptw.phu) on, indicating recent use.

5. The page having its "used in quantum" bit (ptw.phu1) bit on, indicating use since the last post-purging if certain options below are selected.

6. The page having its "modified" bit on.

The four actions that can be taken for each page are:

1. Call the page_writing function to write the page out.

2. Move the main-memory page frame for the page in the used list to the least-recently-used (most replaceable) position.

3. Turn off the used and used-in-quantum (ptw.phm and ptw.phm1. bits.

4. Count the bit in the working set of the process.

The mapping from all sixty-four possible combinations of these attributes into any sub set of the four possible actions is determined by the table "code_tree" in this program. The actions specified by this table in release 5.0 are:

1. Call the page-writing function to write the page out. This is <u>NEVER</u> selected.

2. Move the page to the least-recently-used position of the main memory used list. Done for any page meeting criteria 1 and 3, i.e., in main memory and part of a per-process segment.

3. Turn off the "used" and "used in quantum" bits. This is <u>NEVER</u> selected.

4. Count the page in the process' working set. Done for pages meeting criterion (4), i.e., the used bit is on.

The basic task of the post-purging function is to scan the per-process trace list of pagings performed by a process (see Section VIII for the "per process page trace list") (this function runs in the process it is processing), and to classify the pages involved in the various paging-ins as part of the process' working set or not, and bias the page replacement algorithm in favor of their replacement in various ways.

The post-purge function is implemented in the ALM program post_purge, called by the traffic controller with the traffic control lock not locked. It locks the page table lock at the start of its processing, and unlocks it only at the end of its processing, before returning to the traffic controller.

The post-purge function considers each page reading in the trace list, between the last time the process was post-purged and the current time. It also makes an entry in the trace list, a "scheduling" entry, for use by the "page_trace" command. It considers six attributes of each page in the trace list, and performs up to four potential actions for each page based upon them. These attributes are:

1. The page being in main memory at the time it is seen.

2. The page being on the paging device.

3. The page being part of a per-process (aste.per_process on. segment.

4. The page having its "used bit" (ptw.phu) on, indicating recent use.

5. The page having its "used in quantum" bit (ptw.phu1) bit on, indicating use since the last post-purging if certain options below are selected.

6. The page having its "modified" bit on.

The four actions that can be taken for each page are:

1. Call the page_writing function to write the page out.

2. Move the main-memory page frame for the page in the used list to the least-recently-used (most replaceable) position.

3. Turn off the used and used-in-quantum (ptw.phm and ptw.phm1. bits,

4. Count the bit in the working set of the process.

The mapping from all sixty-four possible combinations of these attributes into any sub set of the four possible actions is determined by the table "code_tree" in this program. The actions specified by this table in release 5.0 are:

1. Call the page-writing function to write the page out. This is <u>NEVER</u> selected.

2. Move the page to the least-recently-used position of the main memory used list. Done for any page meeting criteria 1 and 3, i.e., in main memory and part of a per-process segment.

3. Turn off the "used" and "used in quantum" bits. This is <u>NEVER</u> selected.

4. Count the page in the process' working set. Done for pages meeting criterion (4), i.e., the used bit is on.

There are no installed tools to change the contents of this table, or interpret them.

The post-purge function marks the bit ptw.processed (also known as ptw.er or ptw.pre_paged) in every PTW it processes; it turns off all bits it so turned on before it finishes. Any page it finds with this bit on must already have been processed by this pass; such occurrences are considered evidence of "thrashing", and are counted in the meter sst.thrashing. They indicate the occurrence of a process not being able to keep a page it was using in main memory during one period of eligibility. This action might also turn off PTW error flags by virtue of sharing of this bit, but the worst effect of this would be to cause a process to take an extra page fault to retry and perhaps rediscover a disk or paging device read error.

SECTION X

PERIPHERAL SERVICES OF PAGE CONTROL

This section covers three mechanisms used by the supervisor that can be construed as being part of page control. These three mechanisms are:

1. The facility that temp-wires procedures and their linkage.

2. Paging device reconfiguration.

3. Main memory frame freeing.

These mechanisms do not directly deal with page control objects that are in use. In the first case, no page control objects are dealt with at all; all manipulation of pages is performed by calls upon the services described in Section IX. In the second and third case, objects (CMEs and PDMEs) are threaded into their respective used lists, under the protection of the page table lock. Part of paging device reconfiguration is involved with taking PD records that are in use out of use; this is performed by a page control service described in the previous section ("Paging Device Record Deletion").

PROCEDURE WIRING

(wire_proc)

Many procedures in the Multics supervisor are wired, i.e., may not be removed from main memory. Often this is on account of the fact that they are used during page or traffic control operations, or in processing interrupts. Code invoked in such circumstances may not take page faults, for the taking of page faults may involve page control or traffic recursively, or cause the processor to be lost while a per-processor resource is in use.

Some procedures that may not take page faults are not invoked often; such procedures include much of the code that implements the reconfiguration, and much of the code of the ARPANET interface. Such procedures cause themselves to become wired when they are invoked, and unwired when they return. This procedure wiring/unwiring function is performed by the program wire_proc.

The program wire_proc does not deal with page control data bases at all; it calls pc_wired$wire_wait and pc_wired$unwire (described under "Forced Segment I/O and Wiring," Section IX) to wire and unwire the segments and parts of segments it deals with. The program wire_proc is not itself wired, and does not deal with the page control environment. The basic task of this program is to multiplex requests to wire the same segment; a table is kept of segments it has wired, in the region "sst.wire_proc_data" in the SST. When a request is made to wire a procedure, a check is made to see if that procedure has already been wired by this mechanism, in which case an entry in this table exists for that segment. If not, an entry is made for the segment, and a call to pc_wired is made to wire the segment. In any case, a counter of processes that have called to wire that segment, kept in the table entry, is incremented. When a process calls to unwire the segment, the counter in the table entry (which must exist) is decremented. If and only if the counter reaches zero, a call is made to pc_wired to unwire the segment. Thus, the segment remains wired from the time the first process calls to wire it until the last process calls to unwire it.

Whenever wire_proc wires or unwires a segment, the region of the appropriate supervisor linkage section that contains the linkage for that segment (if it has any), is wired or unwired as well. The program wire_proc checks that it does not try to wire portions of unpaged segments: this case may occur during initialization, when procedures that call wire_proc, which later become paged (See the Multics Initialization PLM, Order No. AN70) are still unpaged, and in cases of procedures with the "wired" attribute defined for their linkage sections in the MST Header (See "generate_mst" in the System Tools PLM, Order No. AZ03, and AN70).

The operations of the program wire_proc, and its table in the SST (which is defined in wire_proc_data.incl.pl1) are protected by a lock, the cell wpd.temp_w_lock in sst.wire_proc_data. Since wire_proc is used by system initialization in collection 1, before the system locking facility is available, wire_proc locks and unlocks this lock, and waits for its unlocking via explicit calls to "stac" and "stacq", with calls to pxss$addevent, pxss$delevent, and pxss$notify for synchronization. The value of the event ID for the unlocking of this lock is "200000000000"b3, and is stored (by init_sst) in the cell sst.temp_w_event in the SST.

The program wire_proc has two sets of entries, wire_proc/unwire_proc, and wire_me/unwire_me. The first two are very rarely used; their caller provides a pointer to the segment to be wired or unwired. The latter are the common pair; the program (i.e. the segment) that calls these entries is assumed to be the target of the wiring or unwiring, and is wired or unwired accordingly.

PAGING DEVICE RECONFIGURATION

(delete_pd_records)

(See also the discussion in the Multics Reconfiguration PLM, Order No. AN71.)

The storage system provides the ability to remove records of the bulk store paging device from use, and add them back. This facility is made available through the operator "addpage" and "delpage" commands. It is implemented in the procedure "delete_pd_records", a part of system reconfiguration that wires itself (via the procedure-wiring service described earlier in this section) and locks the page table lock (via pmut$lock_ptl, see Section VIII) when it actually deals with in-use page control data bases. This procedure also has several entries called by the ring-1 operator environment (See the Multics Operators' Handbook, Order No. AM81) to deal with unflushed paging devices (See "Post Crash PD Flush", Section IX). Among these are included entries that allow the number of unflushed records to be determined, the unflushed instance of the paging device to be abandoned, and a new instance of the paging device to be created and made active.

Except for the part of record deletion that involves evicting pages occupying PD records to be deleted, none of these operations involve dealing with PD records or their PDMEs that are actually in use. Even the operations that free records simply make them available for use. The operation of evicting pages from regions of the paging device being taken out of use is performed by the entry pc$delete_pd_records, which utilizes the methods and routines of the page control kernel to accomplish this. Thus, the procedure delete_pd_records never concerns itself with PTWs, CMEs, or pages of segments.

The entries that add and delete PD records (add_pd_records, delete_pd_records) are called with the first number and number of PD records to be added/deleted. They wire themselves and lock the page table lock when inspecting the paging device map. They both check the validity of their arguments before so doing. The entry to delete PD records does nothing more (once wired, masked and locked) than call pc$delete_pd_records to delete the records; the entry to add PD records does nothing more than thread entries in the region to be added, clearing them and checking before so doing that they were in fact deconfigured previously (first word = -1). Note that entries can be deconfigured by initialization (the program init_pvt) as well as delete_pd_records. Both procedures invoke a subroutine (check_pd_free_and_using) to scan the changed PD map and compute from scratch the parameters sst.pd_free and sst.pd_using, and update the PDMAP header (see Section VI). They call pc$write_pdmap (see Section IX, "Services for Shutdown/Demounting") to write out the changed map to the bulk store. These routines also change the actual "PAGE" CONFIG card in the Multics configuration deck image to indicate up to five pairs of deleted regions of the paging device. If there are more than five deletions, the non-fatal syserr message "delete_pd_records: page card cannot be generated" is issued, and only the first five placed on the card. The PAGE card image, created by the subroutine "build_page_card", is provided only for the use of the "print_config_deck" (pcd) command. The entries add_pd_records may not be used if an unflushed paging device exists.

The four entries scrap_entire_pd, check_pd, scrap_pd_recs, and enable_pd are for use of the ring-1 initializer operator environment for dealing with unflushed paging devices. None of them deal with active paging devices, and thus, they do not wire the procedure delete_pd_records, or lock the page table lock or mask.

The entry delete_pd_records$scrap_entire_pd is invoked by the ring 1 "force_pd_abandon" command. It scans the PD map of an unflushed paging device for any records still in use, (i.e., unflushed, containing unrepatriated pages). As long as such records exist, the system cannot be brought out of ring 1. This entry marks these records as no longer in use, thereby acknowledging that their repatriation has been deemed to be impossible. This operator command is used when a physical volume has been destroyed, and repatriation of PD records to it has become impossible.

The entry delete_pd_records$scrap_pd_recs is similar to delete_pd_records$scrap_entire_pd, but only the PD records pertaining to one physical volume are "scrapped". This facility is not currently used.

The entry delete_pd_records$check_pd is used by the ring 1 operator environment to determine if there are unflushed (unrepatriated) paging device records on an unflushed paging device. It returns the number of such records, and the number of physical volumes on which they appear (the count of distinct PVT indices in the PDMAP entries). Only if there are no such records may the system be brought up. Such records may be taken out of this state by either repatriation (via accepting the physical volumes from whence they came) or the "force_pd_abandon" command, which scraps them.

The entry delete_pd_records$enable_pd is called by the ring-1 operator environment to initialize a new instance of the paging device and its map. It is called at the time the system leaves ring 1, which can only happen if there are no unflushed records on an unflushed paging device. This facility is only used in the case of an unflushed paging device; its first step is to check that this is the case, and in fact that there are no unflushed records (via a scan of the map). This entry scans the map, zeroing all PDMAP entries that are not marked (by init_pvt) as deconfigured, and threads them into the PD used list (as free entries). This action does not allow them to be used; only the variable sst.pd_using being set to a nonzero value enables the PD allocator. Thus, this threading need not even be performed under the page table lock. When all of these entries have been threaded in, the clock is read which produces the unique time value that identifies the instance of the paging device being created, which will be used at post-crash PD flush time (see Section IX) should the system crash non-recoverably with this instance of the paging device active. The subroutine check_pd_free_and_using, described above under the description of add_pd_records and delete_pd_records, is invoked to set the SST variables sst.pd_free and sst.pd_using, and copy relevant parameters into the PDMAP header. The setting of sst.pd_using actually puts the paging device into use, and enables the PD allocator. The labels of all mounted physical volumes are written out, via calls to fsout_vol (see Section IV). This causes the fact that they were exposed to the new instance of the paging device to be recorded in their labels, for possible later use by the post-crash PD flush. As a final action, the active map is written out to the bulk store (via pc$write_pdmap), and a syserr message issued.

MAIN MEMORY FRAME FREEING

Initialization adds page frames of main memory to the paging pool (i.e., removes them from the deconfigured state in which init_sst creates them) as initialization progresses. Similarly, system reconfiguration adds page frames to the paging pool on behalf of the operator "addmem" and "addmain" commands. This facility is provided by the program freecore, which wires itself (via the procedure-wiring facility described earlier in this section) and masks and locks the page table lock (via pmut$lock_ptl, see Section VIII) when invoked. This procedure never deals with main memory frames that are actually in use; thus, it never deals with PTWs, PDMEs, or pages of segments. It is called with the main memory address (as a page frame index into main memory) of a page frame to be freed; it checks, under the page table lock, that indeed, that page frame is deconfigured before proceeding any further.

The program freecore checks the page frame that is to be added for parity errors (via a call to pmut$check_parity_for_use) prior to adding it, typing a syserr message if a parity error occurred. Otherwise, the main memory frame's CME is threaded into the main memory used list, starting this list if it is the first frame so added. Various CME flags and fields are cleared at this time, and the pointers sst.usedp and sst.wusedp (See "Main Memory Replacement Algorithm," Section V) are set to point to this page frame's CME. Counters and meters in the SST and SCS are updated as well.

SECTION XI

QUOTA MANAGEMENT

Quota (page quota, record quota) is the mechanism by which the consumption of segment storage space is administratively controlled. Each nonzero page of a segment consumes a record or unit of quota. Each page of a segment that is in main memory, whether or not it contains zeros, consumes a record of quota. The consumption of quota is controlled by the existence of quota accounts, possessed by certain directories in the storage system. Every segment in the storage system is said to be charged to some quota account. A quota account consists of two numbers, a quota limit (or "quota" proper) and a "used" (or "records used"), being the sum of all of the quota consumptions of all segments charged to this account.

As page control is responsible for the creation and destruction of pages, page control bears the ultimate responsibility for quota management. When page control creates or destroys a page, not only must the "records used" of the concerned segment be adjusted, but the quota account of the directory against which the segment's records are charged appropriately adjusted. Since page creation happens in the page-reading primitive, and destruction on the page-writing and truncation functions, any quota cell against which any active segment's records are charged must be in wired storage, so that it may be referenced via these functions, which run as part of page control, with the page table lock locked. Each AST entry has room for a quota cell, and thus, only the quota cells in ASTEs of directories bearing quota accounts are actually used (although the "records used" field of each directory which does not have a quota account is maintained as though it did; this allows the "get_quota" (gq) command to be used on such directories to report page record usage). The need to keep these quota cells in wired storage requires that all superior directories of a given segment be active. This is the current reason for this need. For each page creation or destruction, page control chases the chain of ASTEs of superior directories of a segment until an ASTE with a quota account (aste.tqsw on) is found; by definition, this is the quota account to which the segment's records are charged.

There are two classes of record quota, segment quota and directory quota, being for pages of non-directory and directory segments, respectively. Each quota cell in the system (in ASTEs and VTOCEs) has space for both types of quota accounts. A directory may have either or both or neither type of quota account. Page control charges segments' pages against the correct type of quota account, as appropriate. However, when creating a page of a directory, quota checking (i.e., checking the appropriate account to see if the quota limit has been passed) is suppressed (as are all page faults with an effective reference ring of zero). This means that directory quota limiting is essentially not implemented in release 5.0; this is due to the impropriety of signalling record_quota_overflow as a means of conveying the exceeding of such limits to directory control.

Since the checking and adjustment of quota cells by page control is performed under the page table lock, adjustment of quota cells via user command or other storage system action must be protected by the page table lock (although some higher lock could have been devised, one would still have to be wired and masked to lock this lock). Thus, page control provides a procedure, "quotaw" (the "w" is for <u>wired</u>), in bound_page_control, which locks the page table lock and adjusts quota cells. It is given as a parameter the AST entry pointer for either a directory whose quota cells are being adjusted, or in some cases the AST entry pointer for a segment, the quota account against which it is charged having to be located and adjusted. This means that all quota cell adjustment must be performed on active directories only; directory control/segment control ensure that this is the case by activating directories to be involved in quota transactions, and passing pointers to their ASTEs to quotaw. The utility program "quota" in bound_file_system is the user visible interface to quota cell manipulation; it identifies directories given their pathnames, locks them and checks access to manipulate quota, handles "master directory quota," activates directories to be involved in quota transactions (using the "activate" primitive; see "Significance of 'activate'", Section IV), and finally, with the AST locked, passes ASTE pointers to quotaw. Segment control, in the segment truncation primitive, similarly activates the parent directory of a segment being truncated, in order to pass its ASTE pointer to quotaw to adjust the relevant quota cell.

The program quotaw has three general entries, "cu", "sq", and "mq", to change the records-used of a quota account, set the quota limit of a quota account, and "move quota" between a quota account and an inferior quota account (decrease limit of one by a certain amount, incrementing inferior's limit by that much). In all cases, a number of records, a quota type (directory or segment quota), and a pointer to the ASTE of a segment (which is charged against the relative quota account, or the directory ASTE itself) is provided as input. The "mq" entry (move quota) takes another ASTE pointer in addition, being the "inferior" ASTE to which quota is to be moved. All of the entries lock the page table lock (via pmut$lock_ptl, see Section VIII) and loop up the AST parent threads to find the correct quota account, and perform the necessary adjustment. The "cu" entry, (change-used, which is generally used to adjust the records-used number of an account) also supports the function of checking whether or not a contemplated change in records-used is valid; an input switch specifies this. All of the entries return a standard status code.

The program quotaw also has a "side-door" (quotaw$cu_for_pc) which is used by the segment-truncation function (in pc$truncate) and the deactivation-time service (pc$cleanup) to adjust quota cells when these functions destroy (or find zero) pages. This entry is similar to quotaw$cu, except that it is called with the page table lock locked, and the process wired and masked, and returns with these circumstances prevailing as well.

SECTION XII

RING ZERO VOLUME MANAGEMENT

INTRODUCTION AND OVERVIEW

Volume management concerns itself with the relation between physical volumes and logical volumes, and between physical volumes and disk drives. It is the responsibility of volume management to ensure the integrity of information upon a given physical or logical volume, and to perform the necessary binding and unbinding operations in the supervisor when volumes are mounted and demounted.

Volume management, as described in these sections, does not concern itself with the operator interface for mounting and demounting, nor the completeness or registration of logical volumes.

There are four sections in this portion of the book:

Section XII	Introduction and Overview
Section XIII	Data Bases of Ring 0 Volume Management
Section XIV	Operations of Ring 0 Volume Management
Section XV	Interaction of the Physical Volume Salvager with the Storage System

Many of the operations that may be considered part of volume management, such as segment moving and physical volume assignment, are covered in Section IV.

Unlike the other subsystems described in this document, no sections describing functions or services of volume management are provided. The only services provided are the mounting and demounting of physical and logical volumes, and the determination of whether or not a given physical or logical volume is in fact mounted. There are no lower-level mechanisms to speak of. Thus, the functions and services of ring zero volume management are placed together under the section "Operations of Ring Zero Volume Management."

CONCEPTS

A physical volume is a disk pack that is described by registration information maintained by the volume registration package in ring 1. A physical volume is divided into records of 1024 words each. These records may contain pages of segments, or be part of the VTOC (Volume table of contents) of the physical volume, be part of partitions, or be part of the volume header. The VTOC consists of entries (VTOCEs), five to a page, that describe the segments whose pages are on this physical volume, one VTOCE per segment. The partitions are conterminous regions of disk set aside for special use, such as FDUMP images and the syserr log. The volume header, which is at a fixed location on the disk, contains information describing the extent and location of the partitions and VTOC, as well as a map (the volume map) of which records are in use by pages of segments. All of the area not in use by the volume header, VTOC, or partitions is called the paging region of the volume, and it is from here that records are used by pages of segments. Every segment described by a VTOCE on this pack has all of its pages on this pack; no segment has pages on several packs. The most important data item in the volume header is the volume label, or label. This data item contains duplication of the registration information kept for this volume, identifying it, and a history of the last use of this volume by Multics. It is this latter information that allows volumes to be used in a consistent fashion across crashes.

The Root Physical Volume, or RPV is the physical volume that contains the root directory, ">", as one of its segments. It is special-cased by the system in many ways. It is the only volume known to the system at the time it is bootloaded. Another segment on the RPV is the disk table, a ring-1 data base that describes the drives on which all packs were located during the last bootload. From this data base, the ring 1 software can bring other volumes into use at the time the system is brought up.

A logical volume is a user-visible collection of physical volumes, designated as such by the volume registration data in ring 1. With the exception of the RPV, no physical volume may be in use by Multics unless all other physical volumes in the logical volume to which it belongs are also in use. Thus, logical volumes are mounted and demounted as a unit. Each directory in the storage system hierarchy has a unique logical volume on whose physical volume all segments immediately contained in that directory reside. This logical volume is called the son's logical volume of that directory.

The root logical volume, or RLV, is that logical volume of which the RPV is a member. The RLV is the only logical volume that may be partially in use. The RLV is the only logical volume that contains directory segments. Although the segments inferior to any given directory reside on the son's logical volume of that directory, the directories reside on the root logical volume. Operationally, the root logical volume is the only one necessary to bring the system up to ring 4 command level, through answering-service startup. The root physical volume is the only physical volume necessary to bring-the system up to ring-1 command level.

To mount a physical volume is to physically place it on a drive and cycle up that drive. This action is performed by the operator, not by software.

To accept a physical volume is to make the necessary calls to the supervisor, for a drive on which a given physical volume has been mounted, to establish in the supervisor the binding between that drive and the physical volume on it. Critical in this binding is the placement of the 36-bit Physical Volume ID (PVID) read from the label of that physical volume in a table entry (the PVTE) associated with that drive. The descriptions of segments in directory branches are in terms of these physical volume IDs, and VTOC indices. Thus, placing this ID in this table entry indicates that the volume is indeed online, and segments on it may be used (via the process of activation, see Section II.

A logical volume is mounted (or "mounted to the system") when all of the physical volumes in it are mounted and accepted, and calls have been made to the supervisor to establish the presence of this complete logical volume on line. Unless a logical volume is mounted, the system will refuse to honor initiations of segments, segment control calls, and segment faults for segments on physical volumes of that logical volume, even though the physical volumes may have been accepted. It is via this policy that the usage of "incomplete" logical volumes is interdicted. One exception to this rule is the root logical volume.

It is mounted even if it is incomplete; it is mounted as soon as the RPV is accepted in system initialization. The system maintains a table of mounted logical volumes, the LVT, or Logical Volume Table. Each entry in it, or LVTE, describes one mounted logical volume, containing per-logical volume information, as well as the start of a chain of PVTEs of accepted physical volumes in this logical volume. Ring 1 will not make the call to mount a logical volume until it has verified that all physical volumes known (from the volume registration data) in it have been accepted.

The system maintains a table, the Physical Volume Table or PVT, containing information about each accepted physical volume. It has one entry, or PVTE, per each disk drive known to the system. This entry contains information about the physical volume mounted on that drive, including its PVID and Logical Volume ID (LVID) of the logical volume to which it belongs. Parameters about this volume, read in from its volume header at the time it was accepted, that are used by page control and segment control in dealing with segments upon this volume and their pages, are kept in the PVTE. The PVTE also contains information used by page control and the disk DIM describing the physical drive associated with the PVTE, such as its device number and device type.

A logical volume may be mounted to a given process or not, if it is mounted at all. A mounted logical volume is mounted to a given process if it is either a public logical volume, or (a private logical volume) a call has been made by RCP in ring 1 in that process to attach the private logical volume to the process. RCP will allow a private logical volume to be attached to a process pendant on whether or not that process has access to the logical volume, as defined by the ACS (access control segment) for that volume, created by the ring 1 registration software. Unless a logical volume is mounted to a given process, the process will act as though the logical volume were not mounted at all; segment faults and initiations are not honored, and segment control calls may not be made. Thus, only those processes selected by the ACS of a private logical volume may use the segments on it, while all processes may use the segments on a public logical volume, subject to the normal Multics access control mechanism and AIM access control. The table of private logical volumes attached (and therefore mounted) to a process is kept in a region of the KST (Known Segment Table) of a process. This set of logical volumes attached to the process is necessarily a subset of the logical volumes that are mounted (to the system), as kept in the LVT. A logical volume that was attached by a given process may be detached by that process, via a call through RCP in ring 1. When this occurs, the logical volume is no longer mounted to the process and segments on it may no longer be used by this process (a local setfaults operation (see Section II) is performed).

A logical volume may be demounted by calling the supervisor to remove the Logical Volume Table entry for it. This prevents further attachments to the logical volume, but does not stop use of the segments in it until each physical volume in the logical volume is demounted. These calls are made by ring 1 volume management in the initializer process.

A physical volume is demounted by making a call to the supervisor (from ring 1 of the initializer process) to stop all processes from using segments on this volume, deactivate all of these segments, flush VTOC buffers of all information relating to this volume, update the volume header of the volume, and remove information from the PVTE for the drive containing that volume which describes it. This unbinds the volume from the drive. At system shutdown time, all volumes are demounted, the RPV being demounted last. At this time, however, a modified form of deactivation is performed that does not involve freeing AST entries or dealing with AST threads (see Section IV).

PREACCEPTANCE

The RPV is accepted, like all other volumes, before segments on it are available for use. For the RPV, this happens during collection 2 of system initialization. However, the RPV is used prior to this, but not segments on it. All of this activity occurs in the hardcore partition of the RPV, and consists of the running of initialization from the running of the program init_pvt up until the acceptance of the RPV. This may involve a volume salvage of the RPV if it had not been shutdown properly during the last bootload. The hardcore partition exists to satisfy the need for a fixed, usable area, for paging by the supervisor, when the validity of the RPV volume map may not be trusted. (See Section VII for more detail on this.)

The point in collection 1 initialization at which the use of the hardcore partition is established, and thus the first paging in initialization begins, is called the preacceptance of the RPV. The RPV label is read, the partition extents on it determined, and the use of the hardcore partition set up. Global system parameters in the FSDCT, relevant to the success of the last shutdown, are determined from the RPV label, as well as the active/unflushed status of any paging device that must exist.

Between the preacceptance and acceptance of the RPV, no VTOC I/O nor segment faults occur on the RPV. No activations occur, nor is the paging region nor VTOC used at all (except the former by the physical volume salvager). The bit-map for the RPV during this time is not the bit-map from the volume-map of the RPV, but rather a special one fabricated by the preacceptance code. It defines the hardcore partition.

The preacceptance of the RPV is performed in the program init_pvt.

SECTION XIII

DATA BASES OF RING ZERO VOLUME MANAGEMENT

This section describes the detailed structure and function of those supervisor data bases that are used to manage the set of physical and logical volumes known to the supervisor. A large part of the visible interface of volume management, however, is that presented by the ring 1 volume management package, responsible for the operator interface and volume registration functions. The data bases of these functions, in particular the Disk Table, the Logical Volume Attach Table, and the Registration Files, are not described herein.

Some of the critical data bases used by ring zero volume management are not seen in the Multics supervisor at all; they are resident on regions of disks, and are explicitly read in and written out at the times that they must be inspected or modified. These data bases, in the volume header of each pack, reside at fixed record addresses on each pack, given in the include file, disk_pack.incl.pl1. These data bases will be described first.

VOLUME LABEL

The volume label resides on the first Multics record of each storage system physical volume. It is generated by the program init_disk_pack_ (for all volumes except the RPV, in which case it is generated by the program init_empty_root) in the ring 1 volume management environment. It is read in at the time a volume is accepted, and written out at the time it is demounted. It is also written out at the time it is accepted, to indicate that the volume was not shut down, until it is written out at the time it is demounted. The label record is divided into five regions, on sector boundaries:

1. GCOS region, sectors 0 to 4 (label.gcos). This region is skipped over entirely, as the Series 6000 GCOS system uses this part of packs as a label area. Avoiding use of this region avoids accidental overwriting of Multics data by labeling a pack under GCOS at a site running both operating systems, and allows some future compatability.

2. Permanent region, sector 5 (label.Multics to label.pad1). This data is per-physical volume information that is never changed. It is written out identically from the copy read in every time the pack label is written. Were it possible to write-protect single sectors, this sector would be so protected at the time the pack was initialized for Multics use. This is permanent identifying information (although some of it is subject to change by the disk rebuilder).

3. Dynamic information, Sector 6 (label.time_mounted to label.pad2). This information relates to the use of this physical volume: the time last mounted, demounted, etc. This information allows the storage system to ensure integrity of data on the physical volume, via its dynamic state.

4. Root information, Sector 7 (label.root to label.pad3). This information is defined only on the root physical volume (RPV) of a hierarchy. It is dynamic information about the entire storage system hierarchy: how successfully if at all it was shut down, and information relative to crash recovery, and bootstrapping the initialization of the directory hierarchy at bootload time.

5. Partition map (sector 10 (octal) (label.parts). A map giving the location and length of any partitions defined on this physical volume. This information is set up at the time that a volume is initialized, and never changed (except by the disk rebuilder).

   The rest of the label record (sectors 11-15, octal) is reserved for future expansion.

Detailed breakdown of the label:

```
dcl 1 label based (labelp) aligned,


    2 gcos (5*64) fixed bin,


    2 Multics char (32) init ("Multics Storage System Volume"),
    2 version fixed bin,
    2 mfg_serial char (32),
    2 pv_name char (32),
    2 lv_name char (32),
    2 pvid bit (36),
    2 lvid bit (36),
    2 root_pvid bit (36),
    2 time_registered fixed bin (71),
    2 n_pv_in_lv fixed bin,
    2 vol_size fixed bin,
    2 vtoc_size fixed bin,
    2 not_used bit (1) unal,
    2 private bit (1) unal,
    2 flagpad bit (34) unal,
    2 max_access_class bit (72),
    2 min_access_class bit (72),
    2 password bit (72),
    2 pad1 (16) fixed bin,
    2 time_mounted fixed bin (71),
    2 time_map_updated fixed bin (71),
    2 time_unmounted fixed bin (71),
    2 time_salvaged fixed bin (71),
    2 time_of_boot fixed bin (71),
    2 pd_time fixed bin (71),
    2 last_pvtx fixed bin,
    2 pad1a fixed bin,
    2 n_bad_tracks fixed bin,
    2 err_hist_size fixed bin,
    2 time_last_dmp(3) fixed bin(71),
    2 dmpr_hd(2) fixed bin,
    2 bk_dmpr_hd(2) fixed bin,
    2 curn_dmpr_item(3) fixed bin,
    2 pad2 (35) fixed bin,
    2 root,
      3 here bit (1),
      3 root_vtocx fixed bin (35),
      3 shutdown_state fixed bin,
      3 pd_active bit (1) aligned,
      3 disk_table_vtocx fixed bin,
      3 disk_table_uid bit (36) aligned,
      3 esd_state fixed bin,
2 pad3 (60) fixed bin,
2 nparts fixed bin,
2 parts (47),
  3 part char (4),
  3 frec fixed bin,
  3 nrec fixed bin,
  3 pad5 fixed bin,
2 pad4 (5*64) fixed bin;
```

label.gcos

Reserved for compatability with the GCOS system. See above.

label.Multics

Contains the character string "Multics Storage System Volume" on every pack. Used for gullibility checks against unlabeled packs, and by resource control to avoid accidental overwriting or disclosure of information on storage system packs.

label.version

Currently must be 1.

label.mfg_serial

Intended to be the manufacturer's serial number for a pack, this is currently set to be physical volume name.

label.pv_name

The physical volume name of the pack.

label.lv_name

Is the logical volume name of the logical volume to which this physical volume belongs.

label.pvid

Is the 36-bit unique ID (PVID) of the physical volume. This same number is contained in the directory branches of all segments contained on this physical volume.

label.lvid

is the 36-bit unique ID (LVID) of the logical volume to which this physical volume belongs. It is contained in all directories for which that logical volume is the sons-logical-volume.

label.root_pvid

is the 36-bit PVID of the RPV of the hierarchy of which this volume is part. This information defines which packs belong to a given hierarchy.

label.time_registered

is currently set to the 52-bit clock time that the volume was initialized for use by the storage system.

label.n_pv_in_lv

is currently not used.

label.vol_size

is the number of Multics records physically available on this volume, regardless of how they are used.

label.vtoc_size

is the number of Multics records used by the Volume Table of Contents (VTOC) and the volume header.

label.private

is "1"b if and only if the logical volume to which this physical volume belongs is a private logical volume.

label.max_access_class
is the maximum AIM access class for segments on the logical volume to which this physical volume belongs. No segments of higher access class (in the AIM sense) can be allocated on that logical volume.

label.min_access_class
is the minimum AIM access class for that logical volume.

label.password
is currently not used.

label.time_mounted
is the last time that this physical volume was accepted by the supervisor.

label.time_map_updated
is the last time at which the label of this physical volume was written. Please note that this name is very misleading; this extremely important quantity, which determines the need to volume salvage (see Section XIV) should be thought of as "label.time_label_written".

label.time_unmounted
is the last time that this volume was demounted, including for shutdown.

label.time_salvaged
is the time that the label was last written out at the completion of processing of this volume by the physical volume salvager.

label.time_of_boot
is the time recorded as "time of bootload" for the system (in the FSDCT) for the last Multics bootload that accepted this volume.

label.pd_time
is the time ("paging device time") identifying the last instance of the paging device to which this physical volume was exposed. By comparing this paging device time to that of an unflushed paging device, repatriation of records may be accomplished. (See "Post-Crash PD Flush" in Section IX).

label.last_pvtx
is the physical volume table index (PVTX) of the drive on which this volume resided at the last time it was accepted by the storage system. By comparing this value with that in paging-device map entries in the PDMAP of the instance of the paging device identified by label.pd_time, repatriation of records may be accomplished. (See "Post-Crash PD Flush" in Section IX.)

label.n_bad_tracks
not currently used.

label.err_hist_size
not currently used.

label.time_last_dmp
reserved for the physical volume dumper.

label.dmpr_hd
reserved for the physical volume dumper.

label.bk_dmpr_hd
reserved for the physical volume dumper.

label.curn_dmpr_item
reserved for the physical volume dumper

label.root
substructure covering the "root information" in the label.

label.here
identifies this physical volume as the RPV of a hierarchy (although other tests will suffice).

label.root_vtocx
is the index in the VTOC of this pack of the directory ">".

label.shutdown_state
is set to various values during the course of shutdown. It is essentially obsolete.

label.pd_active
is "1"b if the system has an active <u>or</u> unflushed paging device. If, at bootload time, when the RPV is interrogated, this bit is on, the system has an unflushed paging device.

label.disk_table_vtocx
is the index in the VTOC of this pack of the segment ">disk_table". Reserved for future use.

label.disk_table_uid
is the unique segment ID of the segment ">disk_table". Reserved for future use.

label.esd_state
is set to zero by the stages of normal shutdown, and to nonzero values by the stages of emergency shutdown. The nonzero value of this variable at the time the RPV is first inspected during a bootload implies that the previous bootload had a successful emergency shutdown. This triggers RPV salvage to collect pages of RPV parasite segments. (See Section VII.)

label.nparts
is the number of partitions on this volume.

label.parts
is an array defining the partitions on this volume. The number of valid entries is given by label.nparts.

label.parts.part
is the four-character ASCII name of a partition.

label.parts.frec
is the first record number on this pack of the partition.

label.parts.nrec
is the number of Multics records used by this partition.

<u>VOLUME MAP</u>

The volume map details which records of the paging region of a physical volume are in use. Although this information may be derived from analysis of every VTOCE on the pack, it is duplicated in the volume map so that record allocations can be performed by page control without inspection of every VTOCE on the pack. If a pack is not shut down properly, this information is considered to be wholly invalid, and is reconstructed by the physical volume salvager via inspection of every VTOCE on the pack. When a volume is accepted, the information in the volume map is copied into the free-store bit-map (see "Disk Record Allocation" in Section VIII) for the drive on which the pack is mounted. It is written back to the volume map on the disk at the time the volume is successfully demounted. The information in the header of the volume map is copied to and from the so-called "fsmap parameters" (see Section VI) in the PVTE for that drive.

```
dcl 1 vol_map based (vol_mapp) aligned,


     2 n_rec fixed bin(17),
     2 base_add fixed bin(17),
     2 n_free_rec fixed bin(17),
     2 bit_map_n_words fixed bin(17),
     2 pad (60) bit(36),
     2 bit_map (3*1024 - 64) bit(36) ;
```

vol_map.n_rec
: is the number of records in the paging region of the pack, and hence, the number of records represented by the volume map.

vol_map.base_add
: is the Multics record number of the first record of the paging region of the pack, and thus the record number of the first bit in the volume map.

vol_map.n_free_rec
: is the number of records in the paging region of the pack which are not allocated. It should be equal to the number of bits which are "1"b in the volume map.

vol_map.bit_map_n_words
: is the number of words in the volume map's bit map. If the number of records in the paging region is not a multiple of 32, the last bits of the last word (the "fsmap tail") will be "0"b, but are <u>not</u> considered part of the bit map.

vol_map.bit_map
: is the array of words that constitute the bit map described by the parameters just described. Neither the first bit nor the last three bits of each word are used, being "0"b in all cases. This leaves 32 bits per word, representing 32 Multics records in each word of the bit map. The value "1"b indicates a free record, and "0"b indicates a record in use.

The volume map is considered to be wholly invalid between the time that a physical volume is accepted and the time that it is successfully shut down or salvaged (see Section XIV).

<u>VTOC HEADER</u>

The VTOC header describes the extent, and global parameters, of the VTOC of a pack. These parameters are copied into the PVTE for the drive on which the pack is mounted at the time it is accepted, and update to the VTOC header from there every time the label is written out. Like the volume header, it is considered wholly invalid (at least the dynamic parameters therein) from the time the volume is accepted to the time the volume is successfully demounted, and must be reconstructed by the physical volume salvager if the volume is accepted without having been shut down.

vtoc_header.version
: currently must be 1.

vtoc_header.n_vtoce
: is the number of VTOC entries (VTOCEs) in the VTOC of this pack, used or free. This is constant, modulo the disk rebuilder.

vtoc_header.vtoc_last_recno
is the Multics record number of the last record occupied by the VTOC of this pack. The first record is currently a constant VTOC_ORIGIN, defined in disk_pack.incl.pl1.

vtoc_header.first_free_vtocx
is the VTOC index of the VTOCE on this pack which is the first in the free chain. This index is maintained in the PVT by the VTOC manager while the pack is in use.

The rest of the VTOC header record is reserved for the physical volume dumper.

BAD TRACK LIST

This information is not currently maintained by Multics.

FSDCT

The letters "FSDCT" stand for "file system device configuration table." This name is largely historical, for the segment that contains free-store bit maps and per-hierarchy information, and has ceased to have any significance.

The FSDCT contains two distinct regions. The FSDCT header contains global data about the state of ring zero volume management. Much of it is derived from the RPV label at the time the RPV is preaccepted during collection 1. Much of it is derived from CONFIG cards, and much of it is written out to the RPV label at various stages of shutdown. It defines the state of shutdown, and the state of ring zero with respect to volume management.

The region of the FSDCT beyond the header consists of the bit maps for disk record allocation for each drive. One region is allocated for each drive, and the volume map bit map from each physical volume is copied in at the time that the volume is accepted. The relative offset of the bit map for each region is defined by the field pvte.fsmap_rel in the fsmap parameters in the PVTE for that drive. The FSDCT is a pageable data base; the withdrawal of disk records from it at page fault time is accomplished via an esoteric maneuver described fully in Section VIII.

The following include file and discussion describe the FSDCT header.

fsdct.shutdown_state
is zero while Multics is running, and set to various nonzero values during normal and emergency shutdown. It is updated to the field label.root.shutdown_state each time the label of the RPV is written out.

fsdct.oos_dir
is obsolete.

fsdct.esd_state
is zero while Multics is running, and set to various nonzero values during emergency shutdown. It is updated to the field label.root.esd_state in the label of the RPV each time the label of the RPV is written out.

fsdct.prev_shutdown_state
is the value of the label.root.shutdown_state in the label of the RPV at the time that the RPV is preaccepted during collection 1 initialization. Thus, it describes the shutdown state of the previous bootload of this hierarchy.

fsdct.prev_esd_state
is the value of label.root.esd_state in the label of the RPV at the time that the RPV is preaccepted during collection 1 initialization. Thus, it tells whether or not this hierarchy last witnessed a successful emergency shutdown.

fsdct.rpvs_requested
is set to "1"b if the operator issued a BOOT RPVS request to boot the system, requesting an RPV salvage (RPVS).

fsdct.root_lvid
is the 36-bit Logical Volume ID (LVID) of the RLV of this hierarchy.

fsdct.root_pvid
is the 36-bit Physical Volume ID (PVID) of the RPV of this hierarchy.

fsdct.root_pvtx
is the physical volume table index (PVTX) of the drive on which the root physical volume (RPV) is mounted. This value is duplicated for various functions in the SST, as sst.root_pvtx. It is derived from the ROOT CONFIG card.

fsdct.root_vtocx
is the index in the VTOC of the RPV of the directory ">", copied from label.root.root_vtocx on the RPV label.

fsdct.rlv_needs_salv
is "1"b if a volume of the RLV needed a salvage at the time it was accepted, and was salvaged. This bit informs the operator interface that a hierarchy salvage of selected directories must be performed at system startup time. (This is because all directories reside on the RLV, and the fact that some volumes of it were not properly shut down may indicate that some directories were damaged.)

fsdct.n_volumes
is not used.

fsdct.dump_part_pvtx
is the physical volume table index (PVTX) of the drive on which the volume with the system's DUMP (BOS FDUMP) partition exists. This drive is selected by the PART DUMP card in the CONFIG deck. It is zero if there is no DUMP partition.

fsdct.dump_part_frec
is the first record number of the DUMP partition, if one exists, on the pack on the drive selected by fsdct.dump_part_pvtx.

fsdct.syserr_log_pvtx
is the physical volume table index (PVTX) of the drive on which the volume with the system's syserr log partition exists. This drive is selected by the PART LOG card. It is zero if the system is not using syserr logging.

fsdct.syserr_log_frec
is the first record number of the syserr log partition, if one exists, on the pack on the drive selected by fsdct.syserr_log_pvtx.

fsdct.syserr_log_nrec
is the number of records in the syserr log partition, if one exists, otherwise zero.

fsdct.free
is not used.

fsdct.hc_exists
is obsolete, and is always "1"b.

fsdct.hc_using
is set on during the preacceptance of the RPV in collection 1, and turned off during the acceptance of the RPV in collection 2. It indicates that the system is running totally in the hardcore partition.

fsdct.hcp_frec
is the first record number on the RPV of the hardcore partition. The RPV must have a hardcore partition defined on it. This number is obtained from the RPV label during preacceptance of the RPV.

fsdct.disk_table_vtocx
is the index in the VTOC of the RPV of the VTOCE describing the segment ">disk_table". It is read in from the root area of the RPV label, but is not now used.

fsdct.disk_table_uid
is the unique segment ID of the segment ">disk_table". Not now used.

fsdct.pd_active
is "1"b if and only if the system has an active paging device. It is set during RPV preacceptance in collection 1, and by shutdown, and is managed dynamically by the cross-bootload paging device management policies. (See Section IX for a discussion of active and unflushed paging devices.)

fsdct.rpv_needs_salv
is set to "1"b during RPV preacceptance if the RPV was not properly shut down during the last bootload. This triggers an RPV salvage later.

fsdct.pd_unflushed
is set to "1"b if and only if the system has an unflushed paging device (see Section IX.)

fsdct.pd_time
is the paging device time identifying the instance of the paging device to which this hierarchy was last exposed. If this bootload never had an unflushed paging device, this is the same as fsdct.time_of_bootload. If the paging device is unflushed, this variable has the value of the variable fsdct.time_of_bootload from the bootload during which that instance of the paging device was active. Otherwise, if the paging device was dynamically enabled during this bootload, this is the time at which that was done. (See Section IX, "Post-Crash PD Flush.")

fsdct.old_root_pvtx
is the value of the cell label.last_pvtx in the label of the RPV. It is used to repatriate RPV pages during acceptance of the RPV. (See Section IX, "Post-Crash PD Flush.")

fsdct.maps
is (*is*, not *contains*) the first word of the bit-map region of the FSDCT.

PHYSICAL VOLUME TABLE (PVT)

The physical volume table, or PVT, is the single most important data base of ring-zero volume management. It contains an entry, or PVTE, for each disk drive known to the system (including so-called "I/O drives"). It also has an entry for the bulk store subsystem (at the end) if one exists, as this is required by page control. The information in the PVTE for each drive describes information needed by the disk DIM to describe that drive with respect to the former's data bases. This includes the device number and subsystem name, as well as the device type. This information stays constant in each PVTE. The PVTE, however, also is filled in with information about the volume mounted on the corresponding drive at the time that such volume is accepted. This information consists of the quantities from the volume's volume header, specifically the VTOC header and volume map. This data is used by segment control and page control to manage the VTOC and the free store bit-map of the volume. Included in the PVTE is also data that describes a region of the FSDCT which is used as the bit-map for each volume mounted on that drive. This information is permanent. The specific parameters for whatever bit-map may be there as a given volume is used is not permanent. The PVT is a paged, wired, deciduous segment, which is used by page control, and thus must not be pageable.

```
dcl 1 pvt based (pvtp) aligned,

    2 n_entries fixed bin (17),
    2 max_n_entries fixed bin (17),
    2 n_in_use fixed bin (17),
    2 rwun_pvtx fixed bin,
    2 pad (4) bit (36),

    2 array (0 refer (pvt.n_entries)) like pvte;

dcl 1 pvte based (pvtep) aligned,

    2 pvid bit (36),

    2 lvid bit (36),

    2 dmpr_in_use (3) bit (1) unaligned,
    2 pad3 bit (24) unaligned,
    2 brother_pvtx fixed bin (8) unaligned,

    2 devname char (4),

    (2 device_type fixed bin (8),
    2 logical_area_number fixed bin (8),
    2 used bit (1),
    2 storage_system bit (1),
    2 permanent bit (1),
    2 testing bit (1),
    2 being_mounted bit (1),
    2 being_demounted bit (1),
    2 check_read_incomplete bit (1),
    2 device_inoperative bit (1),
    2 rpv bit (1),
    2 paging_device bit (1),
    2 salv_required bit (1),
    2 being_demounted2 bit (1),
    2 vol_trouble bit (1),
    2 vacating bit (1),
     2 pad bit (4),


     2 first_free_vtocx fixed bin (17),
     2 n_free_vtoce fixed bin (17),


     2 vtoc_size fixed bin (17),
     2 vtoc_segno fixed bin (17),


     2 fsmap_rel bit (18),
     2 bad_addrs_consecutive fixed bin (17),
     2 dbmrp (2) bit (18)) unaligned,


     2 curwd bit (18),
     2 wdinc bit (18),
     2 temp fixed bin,
     2 baseadd fixed bin,
     2 tablen bit (18) unaligned,
     2 tablen_allocation fixed bin (17) unaligned,
     2 nleft fixed bin,
     2 relct fixed bin,
     2 totrec fixed bin,


     2 dim_info bit (36),


     2 curn_dmpr_vtocx (3) fixed bin unaligned,
     2 n_vtoce fixed bin unaligned;
```

pvt.n_entries
is the number of entries, used or otherwise, in the PVT array.

pvt.max_n_entries
is the same as pvt.n_entries.

pvt.n_in_use
is number of entries corresponding to accepted volumes.

pvt.rwun_pvtx
is the PVT index of a drive (only one may be in this state at a time) expecting an interrupt from the I/O interfacer for cycling down the drive at demount time.

pvt.array
is the array of PVTEs.

pvte.pvid
is the 36-bit Physical Volume ID (PVID) of the accepted volume mounted on this drive, zero if none.

pvte.lvid
is the 36-bit logical volume ID (LVID) of the logical volume to which the accepted volume on this drive belongs, zero if none. As pvte.pvid, this parameter is read in from the volume label at acceptance time.

pvte.dmpr_in_use
is reserved for the physical volume dumper.

pvte.brother_pvtx

is the PVT index of the next volume in the chain of physical volumes belonging to the same logical volume as the one to which the accepted volume on this drive belongs.

pvte.devname

is the four-character ASCII name of the disk subsystem to which this drive belongs.

pvte.device_type

is the hardware device type, as defined in fs_dev_types.incl.pl1, of this disk drive.

pvte.logical_area_number

is the hardware drive number of this disk drive.

pvte.used

is "1"b if and only if there is an accepted volume on this drive. It is _off_ in the PVTE of the RPV until the RPV has been accepted.

pvte.storage_system

is "1"b for a drive that is not an "I/O drive" defined by a "UDSK" CONFIG card.

pvte.permanent

is "1"b for a drive designated by a PART card, and is also "1"b for the RPV. No pack except the one mounted there at bootload time may ever be mounted on this drive during this bootload.

pvte.testing

is set to "1"b by the program read_disk (see Section XIV) before a special call is made to disk_control. This bit tells the disk_dim interrupt side to set pvte.device_inoperative according to the relative success of a "request status" operation on this drive. Disk control turns off this bit when the latter bit has been set.

pvte.being_mounted

is "1"b during the acceptance of a volume on this drive. Primarily informational.

pvte.being_demounted

is set to "1"b at the start of the demount procedure for a volume on this drive. Prevents activations of segments on this volume. (See Sections IV and XIV.)

pvte.being_demounted2

is set to "1"b during the latter part of the demount procedure for a volume on this drive. Prevents VTOC I/O from being initiated. (See Sections IV and XIV.)

pvte.check_read_incomplete

causes page control to store special patterns into core frames into which records of this volume will be read, and check for their presence at the posting of the operation. There is no way to turn this feature on other than patching this bit.

pvte.device_inoperative

is used by the program read_disk, along with the bit pvte.testing, to determine if a drive is operative. (See pvte.testing, above, and Section XIV.)

pvte.rpv

is "1"b in the PVTE of the RPV.

pvte.paging_device

is "1"b in the PVTE of the bulk store subsystem, required by page control to perform abs-seg I/O on the PDMAP.

pvte.salv_required

is set to "1"b during the acceptance of a volume if it was not properly shutdown during its previous use, and thus required and received a volume salvage.

pvte.vol_trouble

is set by various recovery procedures (and ESD) if there is reason to believe that an operation upon the VTOC of a volume is interrupted in such a way that the volume is inconsistent, and will require a volume salvage at some time. This bit being on causes the volume to be shut down in such a way (at the time it is demounted) that it will appear that it was not properly shut down, the next time it is accepted, and thus require and receive a volume salvage.

pvte.vacating

inhibits VTOC allocation (segment creation) upon this physical volume. It is used by the on-line physical volume utility, sweep_pv (see the Multics Operators' Handbook, Order No. AM81, and Section IV, "Segment Control Services" for sweep_pv).

pvte.first_free_vtoc

is the index, in the VTOC of the physical volume accepted on this drive, of the VTOCE that is the head of the free VTOCE chain for this volume. It is maintained by the VTOC manager (see Section III), and copied to and from vtoc_header.first_free_vtocx and acceptance and demount time, respectively.

pvte.n_free_vtoce

is the number of free VTOCEs in the VTOC of the physical volume accepted on this drive. It is maintained by the VTOC manager, and copied to and from vtoc_header.n_free_vtoce at acceptance and demount time, respectively.

pvte.vtoc_size

is the number of Multics records in the VTOC and volume header of the physical volume accepted on this drive. Read in at acceptance time from label.vtoc_size.

pvte.vtoc_segno

is a temporary used by the physical volume salvager.

pvte.fsmap_rel

an "fsmap parameter," is the relative offset into the FSDCT of the region allocated for bit-maps for volumes on this drive.

pvte.bad_addrs_consecutive

is not used.

pvte.dbmrp

is reserved for the physical volume dumper.

pvte.tablen_allocation

an "fsmap parameter," is the length, in words, of the region in the FSDCT allocated for bit-maps for volumes on this drive.

pvte.curwd
pvte.wdinc
pvte.temp
pvte.baseadd
pvte.tablen
pvte.nleft
pvte.relct

are the "fsmap parameters," copies of information in the volume map of the physical volume mounted here, and information needed by and maintained by the free-store allocation algorithm. These fields are described in the PVTE writeup in Section VI.

pvte.dim_info
is information stored by disk DIM initialization for this drive, which the disk DIM needs to perform address computations on this drive, and identify its subsystem.

pvte.curn_dumper_vtocx
is reserved for the physical volume dumper.

pvte.n_vtoce
is the number of VTOCEs, free or used, in the VTOC of the physical volume accepted on this drive.

LOGICAL VOLUME TABLE (LVT)

The logical volume table (LVT) is used to describe all mounted logical volumes. It contains all per-logical-volume data for such logical volumes, and contains threads of the PVTEs of accepted physical volumes that are members of each logical volume. The logical volume ID, however, is duplicated in each PVTE for physical volumes in that logical volume. This enables the segment creation function to operate without a lock. (See Section IV for a description of this activity.) The LVT is a pageable segment, used at segment creation and segment moving time, as well as the time that logical volumes are mounted and demounted (see Section XII).

The LVT contains an entry, a LVTE, for each mounted logical volume. The LVTE for the RLV is set up during initialization (collection 2). The LVTEs for other volumes are set up at the time that they are mounted. The LVT also contains a hash table, hashing LVTEs by their LVIDs of the logical volumes that they describe.

```
dcl 1 lvt aligned based (lvtp),
    2 max_lvtex fixed bin (17),
    2 high_water_lvtex fixed bin (17),
    2 free_lvtep ptr,
    2 pad1 (4) bit (36),
    2 ht (0:63) ptr unal,
    2 lvtes (1:1 refer (lvt.max_lvtex)) like lvte;
```

```
dcl 1 lvte aligned based (lvtep),
    2 lvtep ptr unaligned,
    2 pvtex fixed bin (17),
    2 lvid bit (36),
    2 access_class aligned,
      3 min bit (72),
      3 max bit (72),
    2 flags unaligned,
      3 public bit (1),
      3 read_only bit (1),
      3 pad bit (16),
      3 cycle_pvtx fixed bin (17);
```

lvt.max_lvtex
is the index of the highest-indexed LVTE that can ever exist in this LVT, as defined by the size of the LVT segment.

lvt.high_water_lvtex
is the highest LVT index that was ever used in this bootload. This is a meter.

lvt.free_lvtep
is a pointer to the first in a list of free LVTEs. As they are created as needed, this list is non-empty only if LVTEs have been freed.

lvte.ht
is a hash table, containing pointers to the first LVTEs in the hash threads of each hash equivalence class.

lvt.lvtes
is the array of LVTEs.

lvte.lvtep
for an LVTE in use, is the pointer to the next LVTE in the same LVID hash equivalence class as this one, null if this is the last one. For a free LVTE, it is a pointer to the next LVTE in the chain of free LVTEs, null if this is the last one.

lvte.pvtex
is the PVT index of the first PVTE in the chain of PVTEs for drives containing physical volumes belonging to this logical volume. This chain is threaded through the PVTEs as pvte.brother_pvtx. Zero marks the end of the chain.

lvte.cycle_pvtx
is used by the segment creation function of segment control (see Section IV) to allocate VTOCEs in the logical volume. See that description for its use.

lvte.lvid
is the logical volume ID (LVID) of this logical volume.

lvte.access_class
describes the AIM access class limits of the logical volume.

lvte.public
is "1"b for a public logical volume, "0"b for a private one.

lvte.read_only
is reserved.

PVT HOLD TABLE

The PVT hold table resides in the static section of the program get_pvtx. It is a table of process IDs of processes that start operations on a given physical volume that requires more than one call to the VTOC manager, or a call to the VTOC manager and an action upon the bit-map of the volume. The table consists of an array of marks made by such processes, each mark consisting of the catenation of part of the process' process ID and the PVT index of the volume being modified. These marks are removed when the inconsistent operation is finished.

The purpose of this table is to prevent the volume from being demounted while such an operation is in progress. No process may make a mark in this table if a demount operation has started for a volume on which an operation was about to begin (pvte.being_demounted prevents this). Similarly, the demounting procedure demount_pv will wait for all marks in this table relative to a particular physical volume to vanish before the demount procedure can continue.

If a process suffers a crawlout at such a time that it had made a mark in this table, and thus left a volume in an inconsistent state, not only is its mark or marks removed from the table, but that volume is scheduled for a salvage via setting of the bit pvte.vol_trouble (see earlier description of this bit). This is also the case if an ESD occurs after a system crash at which time processes had marks in this table.

The segment mover marks two volumes at a time in this way.

The PVT hold table can be located, for crash analysis and debugging purposes, from the sppointer sst.pvthtp.

SECTION XIV

OPERATIONS OF RING-0 VOLUME MANAGEMENT

ACCEPTANCE OF PHYSICAL VOLUMES

The acceptance of physical volumes is the most fundamental and important operation of ring zero volume management. This service is provided for ring 1 volume management, which controls the operator and cross-process interface, at the time that the latter wishes to make a logical volume available for use. All of the physical volumes in a logical volume are accepted by ring 1 volume management before the logical volume is declared to be mounted (entered in the LVT). The main procedure of volume acceptance is accept_fs_disk.

Physical volumes are accepted by calling initializer_gate_$accept_fs_disk, with the PVT index of the drive on which the physical volume to be accepted is mounted. The ring 1 volume management and registration package ensures that the volume on the drive is the correct one requested by the operator or requesting processes. Ring zero volume management assumes that it is correct, and derives all data from the label of that volume. The RPV is accepted in a special fashion during collection 2 of bootloading; the operator, by issuing the BOOT command, and by use of the ROOT CONFIG card, has assured that the drive described by that card is the legitimate RPV. Thus, this physical volume is accepted automatically in ring zero without having been validated by ring 1.

The essence of physical volume acceptance is to initialize the PVTE for the drive on which the volume being accepted is mounted with data from the label, VTOC header, and volume map of that volume, and mark the PVTE as belonging to a volume in use. This latter step is the last step. Thus, there are no race conditions in determination of whether or not this volume is actually accepted. Since segment creation is driven off the logical volume table, and initiation checks there as well, it is only in the case of non-RPV volumes of the RLV that there is even an issue, for only in this case is there a LVT entry before all PVT entries are set up.

An auxiliary task of physical volume acceptance is to copy the volume map into the region allocated in the FSDCT for bit maps of volumes mounted on that drive. This function is performed in the procedure load_vol_map, which constructs a PTW-level abs-seg to read the volume map from the disk. This procedure also takes responsibility for reading the VTOC header (via the same abs-seg) and initializing PVTE parameters derived from the latter from it. In the case of the loading of the volume map of the RPV, page control activity is halted, via wiring, masking, and locking the page table lock, while the volume map is being copied. This is because the bit-map region of the FSDCT for the RPV will contain the bit-map of the hardcore partition at this time, and will actually be in use at that time. Although all pages of the supervisor should be withdrawn at that stage, and thus no activity on this bit map should take place during the copy, this policy assures that none in fact will take place. This policy dates from a time before all supervisor pages were prewithdrawn. The program load_vol_map also takes responsibility for filling in these PVTE parameters derived from the volume map.

It is also the responsibility of physical volume acceptance to determine if a physical volume needs salvaging, and call the physical volume salvager if so. A physical volume needs salvaging if it was in use, not properly shut down, and not salvaged since it was used. The volume map and VTOC may not be used validly unless this salvage is performed. Each time that a physical volume is accepted, the label is written out at the end of the acceptance procedure (via a call to fsout_vol), which sets label.time_map_updated to the current time. Each time that a physical volume is properly demounted (including shutdown), the label is written out, but this time, setting both label.time_map_updated and label.time_unmounted to the current time. Thus, if an attempt is made to accept a physical volume for which the value of label.time_map_updated and the value of label.time_unmounted are not equal, then this volume was not properly shut down. If, however, the volume has been salvaged since it was last used, it need not be salvaged again. The volume salvager writes out the label with label.time_map_updated and label.time_salvaged equal to the current time. The equality of these two label fields implies the completion of a volume salvage since last use. The procedure accept_fs_disk makes these checks for all volumes except the RPV; init_pvt, at RPV preacceptance time, makes these checks for the RPV.

The automatic salvaging of volumes during acceptance includes the repatriation of pages from that volume left on the paging device during the previous (or earlier) bootload. This is done in the case where the system has an unflushed paging device, and the physical volume salvager detects that the volume was not previously shut down, and exposed to the system's instance of the paging device. (see Sections IX and XV.)

PHYSICAL VOLUME DEMOUNTING

The demounting of physical volumes involves reversing all of the steps taken at acceptance time, and physically cycling down the disk drive on which a physical volume is mounted. Physical volume demounting is complicated by the fact that at the time that a physical volume is demounted, any number of processes may be using information on that physical volume, and may be depending upon its mounted and accepted status. The problems of demounting are thus two, the flushing of supervisor data bases of all information about the physical volume, and the stopping of processes that are using information on it, in a recoverable way.

The principal goal of demounting is the updating of all information on that physical volume with the latest copies of information resident in the AST, FSDCT, and in frames and records of main memory and paging device. This implies writing back all pages in main memory and paging device to their assigned addresses on that physical volume, and the updating of all VTOCEs for segments on that volume from the AST. These two steps are accomplished by deactivating all segments (see Sections II and IV) from that physical volume which are active at demount time. The VTOC manager's VTOC buffer segment must be flushed of all vtoce-parts from this volume, and all pendent I/O on it awaited. This step, clearly, is performed after the deactivation of all segments on the volume. The volume map, VTOC header, and label of the volume must be updated from the FSDCT and PVTE for the volume.

The procedure that coordinates demounting is demount_pv, also known as the demounter. The final stages of demounting, viz., updating the volume map, VTOC header, and volume label, are performed by fsout_vol, called from demount_pv. It is essential to realize that all volumes are demounted at shutdown time, both emergency and regular. There are only two differences between normal demount and shutdown demount.

1. At normal demount time, the drive containing the volume to be demounted is cycled down via a series of calls to the I/O Interfacer. At shutdown time, no drives are cycled down.

2. At normal demount time, segments are deactivated via a call to "deactivate," the normal segment control deactivation procedure. At shutdown time, explicit calls are made to pc$cleanup and update_vtoce (the two procedures at the heart of deactivation) to avoid dealing with possibly bad AST threads (and to allow deactivation of directories with active inferiors. Directories are all on the RLV, which cannot be demounted via normal demounting).

The demounter begins by turning on the bit pvte.being_demounted, and waiting for all processes engaged in multistep operations on this volume to finish. Turning on this bit, as explained below under "Demount Protection," prevents the inception of any new multistep operations on this volume after the time it is turned on. The demounter then locks the AST and deactivates (or, in the shutdown case, simulates deactivation of) all segments on this volume. This deactivation is performed under the AST lock; all processes seeking to activate a segment check the bit pvte.being_demounted at such time as they acquire the AST lock. Thus, since no process except that of the demounter holds the AST lock at this deactivation time, any process except that of the demounter holds the AST lock at this deactivation time, any process attempting to activate a segment, that did not succeed in fully activating it before the demounter acquired the AST lock, will acquire the AST lock after the demounter, and thus find the bit pvte.being_demounted <u>on</u>, and fail to activate the segment. Therefore, the deactivation of all segments on the volume is total and irreversible; it deactivates all segments that were active when it acquired the lock, and no segments (on that volume) will be activated after it releases it. The deactivation purges all data relevant to the volume being demounted from the AST and from page control, and makes the copies of all segments on the disk, and all VTOCEs accurate. This is what is normally done by deactivation (see Section IV); it is simply being performed here for all active segments on the volume.

The second phase of the demounter is the cessation of VTOC I/O activity for the volume. This begins by setting the bit pvte.being_demounted2, which prevents the inception of any VTOC I/O activity for the volume not already under way. As the deactivation phase of demounting starts a great deal of VTOC I/O activity for the volume, which does not complete in that phase, this phase must follow the deactivation phase. A call is made to the VTOC manager (vtoc_man$cleanup_pv, see Section III) to await all I/O in progress for vtoce-parts of this volume, and make a final attempt at flushing "hot" vtoce-part buffers (those that have suffered write errors). Before this call returns, all data relevant to the physical volume being demounted will have been flushed by the VTOC manager from its data bases. This call involves the VTOC manager locking its VTOC buffer lock. All other calls to the VTOC manager check the bit pvte.being_demounted2 under the protection of this lock, and return an error code (error_table_$pvid_not_found) if the PVTE of a volume specified to it has it on. Therefore, all VTOC I/O operations underway at the time the demounter acquires the VTOC buffer lock will be awaited to completion by the demounter, and, since any potential operation not under way by then will acquire the lock after the demounter and find pvte.being_demounted2 on, no new operation may be started after the demounter has released the lock. Therefore, the purge of information about the volume is total and irreversible; all VTOC I/O activity is complete for the volume, and no new activity may be started.

The third phase of demounting is performed by fsout_vol, which, in general, updates labels on disks. In the case of a demount, all parameters in the volume map (including the bit map itself), and the VTOC header are updated as well. The cells label.time_unmounted and label.time_map_updated are set to the same value (the current time), which indicates to the next attempt to accept this volume that it was successfully shut down. These policies are explained under "Physical Volume Acceptance" in this section. Once the label has been written out, the parameters in the PVTE for the volume's PVID to fail, and allowing reuse of the drive for acceptance of a (probably different) physical volume.

The final phase of demounting, which is not performed at shutdown time, is the cycling down of the drive on which the volume being demounted is mounted. This is performed via a "hardcore" attachment to the I/O interfacer to issue an "unload" command to the drive. An attachment is made, for the demounter process, via direct calls to the I/O interfacer. The resource control program (RCP) is not involved in any way. A workspace segment is set up by the I/O interfacer, and the procedure fs_unload_disk_interrupt is set up as the interrupt handler for the attachment. A connect is issued to the drive, to execute the "unload" command. The demounter sets a cell (pvt.rwun_pvtx) to the PVT index of the drive to which the "unload" command was issued before issuing the connect, and loops awaiting the zeroing of this cell. The interrupt-side program (fs_unload_disk_interrupt), after making a few checks, zeros this cell upon receipt of IOM status from the unload operation. The use of this single shared cell prohibits the demounting of several volumes in parallel; this fact is enforced by the restriction that only the initializer process can perform demounting. A single shared cell is used because the I/O interfacer provides no facility for its interrupt side to identify a device to a subsystem's interrupt handler in terms known to that subsystem. Thus, as there is no simple way to determine the PVT index of a drive to which an "unload" was issued at interrupt time, a single cell is used.

Demount Protection

The demounter poses to segment control the problem of the validity of PVT indices; a PVT index derived via search of the PVT for a given PVID is valid if and only if pvte.being_demounted was not on (volume was not being demounted) at the exact instant that the PVID was found in the PVT, and remains valid only as long as this is so. By "valid," we mean that use of this PVT index, by page control or VTOC management, will indeed result in a reference to the physical volume whose PVID was sought to determine this PVT index. Thus, a PVT index which was "validly" derived via PVT search can become invalid instantaneously as another process executes the demounter. Thus, without further mechanism, PVT indices would be useless, as they could be invalidated at any time. Mechanisms therefore exist to implement demount protection, via which processes can either ensure or determine the validity of PVT indices at any time.

The simplest of these mechanisms is the "unitary operation" facility provided by the VTOC manager. This can be used by any function that involves only a single interaction with the volume, and that interaction must be via the VTOC manager. Such an operation is the reading of "VTOC attributes" (see Section IV). A single call to the VTOC manager is adequate to supply such information. Another is the allocation of VTOCEs (see "Segment Creation," Section IV), for which exactly one call to the VTOC manager allocates and writes out a VTOCE. Such operations are said to be "unitary;" either the VTOC manager will succeed in performing them totally, or report that the physical volume is not mounted. These operations are made possible by supplying the PVID of a volume on which an interaction is necessary along with a possibly-valid PVT index for the drive on which that volume is (probably) mounted. This PVT index can be obtained via a call to get_pvtx$get_pvtx, which will make a perfunctory check for a being-demounted bit, and return the PVT index of the physical volume (if any) with that PVID. It is no matter that the volume may be demounted (pvte.being_demounted turned on, or fully demounted) after this search has been performed; the VTOC manager will check the PVTE specified by the PVT index supplied against the PVID supplied under the protection of the VTOC buffer lock before commencing any operation. If the PVID does not correspond, or the bit pvte.being_demounted2 is on (the point at which VTOC I/O request inceptions will no longer be honored), the request is refused. If the PVID corresponds, and the bit pvte.being_demounted2 is not on, the demounter cannot proceed, or even turn on this bit, until it acquires the VTOC buffer lock (see the preceding discussion) and cannot complete until the operation that is being requested here has finished (no vtoce-part buffer out_of_service bits are on).

If the operation being requested requires several vtoce-part I/Os, with intervening unlocks of the VTOC buffer lock, the operation may fail in an intermediate state. However, the design of the VTOC manager is such (see "VTOC Manager, General Policies," Section III) that no irreversible action will have been taken until all vtoce-parts are acquired in buffers under the protection of the VTOC buffer lock.

Another form of protection against demounting is provided to those procedures which operate under the protection of the AST lock. This specifically includes segment deactivation. Since the demounter must lock the AST in order to deactivate all segments, and, as shown above, no new segments can be activated after it has finished this activity, any PVT index obtained (under protection of the AST lock) from an ASTE is valid as long as the AST is locked to the process that obtained it, in the same locking. Any process that derives a PVT index by other means (PVT search, for example), while the AST is locked, is ensured of the validity of that PVT index for as long as the AST is locked, provided that pvte.being_demounted was not on at the time that it derived it (shortly before, or after, so long as the check is made with AST locked).

A similar form of protection is provided to the VTOC manager; if an operation is commenced under the protection of the VTOC buffer lock, and pvte.being_demounted2 was determined not to be on shortly after this lock was locked, the demounter cannot acquire the VTOC buffer lock as long as it is held by the current process, and thus the validity of the PVT indices so validated is ensured.

The most general form of demount protection is provided by the "demount protection brackets" implemented by the entries get_pvtx$hold_pvtx and get_pvtx$release_pvtx. Between a call to get_pvtx$hold_pvtx that does not fail (return an indication of demounted or demounting volume) and a call to get_pvtx$release_pvtx with the returned PVT index, by the same process, the volume specified by PVID to get_pvtx$hold_pvtx will not be demounted. The first call places, and the second call removes, a "mark" in the PVT hold table, specifying the process and the PVT index of the volume concerned. The demounter waits for all such marks for a given volume being demounted to be cleared from the PVT hold table as one of the first steps in demounting. To ensure that no new marks for a given physical volume are made once the demounter awaits the removal of all marks for that volume, the bit pvte.being_demounted is turned on before the demounter awaits the removal of these marks. The entry get_pvtx$hold_pvtx will return a failure indication if this bit is on before it makes its mark, and will remove its mark and return a failure indication if this bit is found on after it makes its mark. The entry get_pvtx$drain_pvtx is used by the demounter to await the removal of all marks relative to a given physical volume.

The demount protection brackets are used to "bracket" multistep interactions with a physical volume, protecting the entire interaction against the demounter. When such an operation has commenced, the demounter may not progress in such a way that would invalidate that operation until the operation is over. If a demount is in progress, such an operation may not even begin. Typical multistep volume interactions are truncation and deletion of segments. Truncation involves calling the VTOC manager to write back a VTOCE without certain addresses, followed by the depositing of these addresses (to the FSDCT). Should the volume concerned be demounted between the VTOCE write and the deposition, the deposition would address an invalid volume map in the FSDCT. Similarly, deletion of a segment involves truncation and freeing of a VTOCE; should the volume be demounted between the truncation and the freeing, a zero-length segment would appear on the volume the next time that the volume is accepted. Thus, these multistep operations must be bracketed by calls to get_pvtx$hold_pvtx and get_pvtx$release_pvtx, protecting the volume against demounting, and allowing the PVT index produced by the former to be used validly (without the protection of the AST lock).

Should a process encounter an ansychronous interruption (such as a "crawlout," process termination, or a crash followed by an emergency shutdown) at the time that a volume is "held" by the demount protection bracket mechanism, the procedure verify_lock (in the first two cases, or wired_shutdown in the third) will clear the mark from the PVT hold table, and schedule the volume for later salvage via the setting of the bit pvte.vol_trouble. This will cause later demounting of that volume to write out the label in such a way that it is volume-salvaged the next time that it is accepted.

The segment mover holds two volumes at a time, the two engaged in the segment move.

RING ZERO LOGICAL VOLUME MANAGEMENT

The logical volume is an instrument of convenience used to compensate for the inadequacy of a physical volume, in size, to hold an arbitrary number of segments. As such, the mounting of logical volumes is little more than the acceptance of several physical volumes, and the demounting of several physical volumes. Thus, the mounting and demounting of logical volumes is little more than the preparation and destruction of entries in the logical volume table describing the logical volume. Ring zero logical volume management also consists of the maintenance in the KST of each process of a small table of logical volume IDs (LVID's of private logical volumes mounted to that process.

Other than the setting of per-process (KST) and per-system (LVT) table entries, marking logical volumes as mounted or not mounted to the system or the calling user process, logical volume management provides only two services to the rest of the supervisor:

1. Answering the question of whether or not a given logical volume is mounted to the calling process. For a public logical volume, this is equivalent to whether or not it is mounted at all (to the system). For a private logical volume, it must be mounted to the system and attached to the invoking process. The procedure "mountedp" answers this question in general, given the LVID of a given volume. This code is duplicated in the segment fault handler for efficiency.

2. Providing the head of the PVT chain for a given logical volume, for the segment (VTOCE) creation function, described in Section IV. This service is provided by logical_volume_manager$lvtep, which returns a pointer to the appropriate LVTE, or null if that volume is not mounted to the system. This pointer may be invalidated at any time; the LVIDs of physical volumes as stored in the PVT are cross-checked by the segment creation function to account for this fact.

The logical volume table is manipulated without a lock; this is because only the mount/demount process (the Initializer) may modify it. Processes that search it are aware that the results of searching it may be instantaneously invalidated. Only in the case of segment creation is this an issue; at other times, subsequent calls to the VTOC manager will fail if physical volumes are demounted after a subsequently invalidated logical volume presence is deduced from the LVT. The LVT is managed by the program logical_volume_manager. The entries to add and delete logical volumes from the logical volume table (logical_volume_manager$add and logical_volume_manager$delete, respectively), are called by the ring 1 volume management package in the initializer process, which implements the operator interface, via the gate initializer_gate_. The "add" entry builds the LVTE from information supplied, and threads together the PVT chain of all existent PVTEs with an LVID equal to the LVID of the volume being added to the LVT.

The "delete" entry destroys this thread, and frees the LVTE. An entry exists (logical_volume_manager$add_pv) which adds a PVTE (and thus a physical volume) to an alreadymounted logical volume. This is used by the ring 1 volume management package when other physical volumes of the RLV than the RPV are accepted, at which time the RLV is already mounted, and at the time that new physical volumes are created and accepted while a logical volume is mounted.

The table in the KST (kst.lv) of private logical volume LVIDs is used to answer the question of whether or not a private logical volume is mounted to the process owning the KST. A call to private_logical_volume$connect, from the ring 1 Resource Control Package (RCP), adds an LVID to this table. Before this call is made, RCP validates the caller's access to the logical volume, and the fact that it is mounted to the system (at least immediately before the call is made). This call is made via the gate admin_gate_. The complementary call to private_logical_volume$disconnect removes an LVID from this table. At the time any segment on a private logical volume is initiated in a process, its index in this table is stored in its KST entry (kste.infcount, multiplexed because all directories, the only segments with nonzero inferior counts, are on the RLV, a public volume). At the time that an LVID is removed from a process' KST, a setfaults operation (setfaults$disconnect, see Section II) is performed on each known segment in this process on that logical volume. This causes the immediate revocation of access to that volume for the process, as the segment fault handler checks whether or not a logical volume is mounted to a process (defined) before honoring a segment fault on that volume for that process.

The entry private_logical_volume$lvx exists to answer the question as to whether a given LVID appears in the calling process' KST, i.e., is mounted to that process given that it is mounted to the system (as determined by logical_volume_manager$lvtep).

BOOTSTRAPPING OF LOGICAL VOLUME HIERARCHY (THE RPV)

The system must be booted to command level before the operator can issue commands to cause the acceptance of physical volumes and the mounting of logical volumes. However, the running of the operator software, and the loading of the system library segments into the hierarchy, involves directories in which to put them, and thus the existence of the root logical volume, before these commands can be issued. Thus, it would at first seem that the RLV must be mounted before the system comes up. Mounting of logical volumes automatically by ring zero is undesirable, as it requires that ring zero be informed of the location of these volumes via CONFIG cards, or various inflexible forms of contract based upon configurations during the last bootload. The responsibility of validating labels resets upon the ring 1 volume management package. Thus, the compromise is made that only one physical volume of the root logical volume must be present at bootload time; this volume is the RPV, and the description of its drive via the ROOT CONFIG card constitutes validation of the RPV pack as the RPV by operator. All of the directories needed by bootloading, that already exist, must be on this particular volume of the RLV. Furthermore, the segment used by cross-bootload ring 1 volume management (>disk_table) to specify the location of packs during the last bootload, must be available on this volume, as all volumes are assumed, by covenant with the operator, to assume their positions during the last bootload unless otherwise specified.

All of the directories so needed are either the root directory itself (>) or one of its immediate descendants (>dumps, >system_library_1, or >process_dir_dir.) Thus, by placing the cross-bootload disk configuration segment (>disk_table) in the root directory, the rule can be made that all immediate descendants of the root directory (segments or directories) must be allocated on the RPV. The segment creation function (see Section IV) carries out this policy; any segment or directory created off the root directory can only be created on the RPV. The segment mover will not move such segments off the RPV.

An implication of this policy is that the RLV must be mounted to the system (so that segment creations and segment faults may be honored upon it) while only the RPV is accepted. System initialization causes this to be the case by calling logical_volume_manager$add for the RLV at such a time during initialization that the RPV has been accepted. Ring zero has no notion of the completeness of volumes; any time that a call is made to logical_volume_manager$add, that volume becomes usable, and consists of all of the physical volumes in it accepted at that time. All segment creations will be restricted to those volumes. Thus, all segments created by initialization reside on the RPV.

RPV-only Directories

When the system arrives at ring 1 command level, the RPV is the only physical volume accepted, and the RLV the only logical volume mounted. In order to register other logical volumes, and check their labels, the logical volume registration data base must be present. Thus, the logical volume registration segments used by the ring 1 volume management and registration package must be on the RPV. Rather than put these segments in the root directory, a directory exists (>lv) which has the property, like the root directory, that all of its inferior segments are restricted to allocation upon the RPV. The bit dir.force_rpv in this directory's header (set by set_sons_lvid$set_rpv, see the following discussion), has the same effect upon the segment creation function as creation of an immediate descendant of the root directory.

One peculiarity in this policy exists. Segments created by bootloading in >system_library_1 are not bound to stay on the RPV, and may be subject to segment moving. If the next bootload, which generally deletes all segments in >system_library_1 that appear on the new bootload tape, finds such a segment, which has been segment-moved, it cannot delete it. Initialization renames it in order to load the new one, with a message from make_branches$delete indicative of this fact. Such segments may be deleted by the "ldelete" command by system maintenance personnel, when the system is fully up (the entire RLV accepted).

Cold Boot of the RPV

During a cold system boot, when there is no hierarchy at all, the system must arrive at ring 1 command level before any volume registration commands can be issued. The RPV must be fully initialized and registered before it can be used, but before the system comes up. Therefore, the program init_pvt, when it detects a cold boot situation, "registers" the RPV by generating an LVID and PVID for it based upon the clock value. The program init_empty_root is called in this case, which writes a valid label for the RPV, including in it information placed on the special-format PART cards used in such a circumstance (see the Multics Operators' Handbook, Order No. AM81). The volume map, VTOC header, and VTOC are initialized, using default parameters generated by init_empty_root. The program that initialized the volume map, VTOC header, and VTOC, init_vol_header_, is available in all rings (a deciduous segment, see Section VII), and is used by the ring 1 volume management package to initialize other volumes. It takes as an argument an entry variable, specifying a routine that is used to write to the pack.

The ring 1 volume registration package (at mdx$reregister) constructs the RPV's registration information (as well as the RLV's initial registration information) based upon the information generated by initialization in ring 0 and written to the RPV label.

SONS-LVID SETTING

The directory field dir.sons_lvid is the logical volume ID (LVID) of that logical volume on which all immediately inferior segments to that directory will be allocated; this value is used by the segment creation function to obtain (via logical_volume_manager$lvtep) the head of the PVT chain of that logical volume. This value is also "inherited" as a son's-LVID by all directories created inferior to this directory. In all cases except the case of the creation of a master directory, this quantity is in fact inherited by the directory control directory creation primitive. In the case of a master directory, this value is specified by master directory control.

The sons-LVID of a directory may be changed dynamically, via the set_sons_lvid command (see the Multics Administrators' Manual -- System Administrator, Order No. AK50) if that directory has no immediately inferior segments (but may have inferior directories). This primitive accesses the program set_sons_lvid in ring zero via the gate hphcs_. This program simply changes the sons-LVID field of the directory, and marks it as (implicitly) a master directory, marking the ASTE as well as necessary. This feature is useful to cause process directory segments to be allocated on logical volumes other than the RPV; bootload re-creates >process_dir_dir each bootload, after renaming the old one. Thus, >process_dir_dir (and the initializer's process directory, >pdd>!zzzzzzbBBBBBB), have a son's logical volume of the RLV. Setting the son's-LVID of >process_dir_dir to some other logical volume after the system is up causes newly created process directories to inherit that son's-LVID, rather than the RLV.

RPV-ONLY DIRECTORY SETTING

The program set_sons_lvid also includes an entry, set_sons_lvid$set_rpv, to set the RPV-only bit (dir.rpv_only) for some directory whose son's-LVID is already the root logical volume (RLV). This facility, accessed through the gate hphcs_, is used by the ring 1 volume management package, to force volume registration files in the directory >lv to be on the RPV, so that they will be available in the ring 1 operator environment at bootload time, whether or not any other physical volumes of the RLV have been accepted.

DISK TABLE LOCATION SETTING

A facility exists to store the VTOC index (in the RPV VTOC) and unique ID of the segment >disk_table in the label of the RPV. The ring-0 primitive set_disk_table_loc is called (via the gate initializer_gate_) at the time the ring 1 volume management package is initialized to set this information. It obtains it from the branch of that segment, and stores it via reading and writing the RPV label. This information is placed there for the use of an unimplemented facility whereby BOS SAVE would be able to determine the location of physical volumes by reading the disk table, rather than receiving volume location specifications on individual request lines.

EXPLICIT DISK READING, WRITING, AND TESTING (read_disk)

Volume management provides a utility program (read_disk), which, given a (guaranteed valid) PVT index, record number, and data buffer, reads or writes that record from/to that data buffer. This is accomplished via the use of a PTW-level abs-seg (see Section VII), rdisk_seg. In the reading case (read_disk), a live device address, the record address desired, is placed as a "disk devadd" (see Section VI) in the single used PTW for this segment, and data copied from rdisk_seg to the caller's buffer. In the write case, a nulled "disk devadd" describing the record described is placed in the PTW, and the data copied from the caller's buffer to rdisk_seg. The nulled address prevents the old data from being read in in order to page the new data out. This is relevant performance of this primitive in cases where it is used in a loop (such as volume initialization). After either call, pc$cleanup is used to force the page of rdisk_seg out of main memory (see Section IX, "Deactivation Service"), in the write_disk case, causing the actual write, and guaranteeing its completion to the caller. The PVT index supplied, in either case, is placed in the ASTE for rdisk_seg before the reference to this abs-seg. This selects the drive to be addressed.

The primitives read_disk and write_disk call a special entry in the disk DIM (disk_control$test_drive) to determine if a drive is patently inoperable before attempting to use it via abs-seg (paging) I/O, which would generate disk DIM and page control error messages in that case. This special entry is used by turning on the bit pvte.testing in the PVTE for the drive concerned, calling it, and looping on the bit pvte.testing, waiting for it to be turned off by the disk DIM interrupt side. The DIM issues a "RQS" (Request Status) operation on behalf of this entry, and sets the bit pvte.device_inoperative to report the outcome of this operation. The bit pvte.testing is turned off once pvte.device_inoperative is set appropriately. If this test indicates an inoperative drive, read_disk and write_disk return an appropriate error code, and do not attempt paging I/O on the volume. This testing function is also available explicitly via the entry read_disk$test_drive.

The read_disk and write_disk entry points are used by acceptance and physical volume demounting to read and write labels, VTOC headers, and volume maps (although load_vol_map uses its own abs-segs). These facilities are also available to the ring 1 volume management package via initializer_gate_$read_disk and initializer_gate_$write_disk, to verify labels, and perform volume initializations. As there is only one ASTE for the abs-seg rdisk_seg, all of these activities are confined to the initializer process, or the process performing emergency shutdown.

SECTION XV

INTERACTION OF THE PHYSICAL VOLUME SALVAGER WITH THE STORAGE SYSTEM

This section describes the actions performed by the physical volume salvager as they are relevant to the actions performed and assumptions made by volume management, segment control, and page control. It does not attempt to explain the internal organization of the physical volume salvager, its interface with the rest of the Multics salvager subsystems, or the interpretation of its printed diagnostics. For these details, see the Multics Storage System Salvager PLM, Order No. AN62, and the Multics Operators' Handbook, Order No. AM81.

The physical volume salvager is invoked upon a single physical volume. It may be invoked either by explicit operator command (the salvage_vol Initializer command, see the MOH), or automatically by physical volume acceptance (see Section XIV) if the latter determines that the volume being accepted was not properly demounted during its last use. The physical volume salvager inspects and modifies the label, volume map, VTOC header, and VTOC of a physical volume, using abs-seg I/O. The tasks of the physical volume salvager are two:

1. To make valid a set of assumptions about the VTOC and volume map of a physical volume, on which the proper operation of segment control, page control, and volume management depend. These assumptions are detailed below.

2. To detect and correct random and unexplained damage, due to hardware or software failure, to the VTOC and volume header of a volume.

The first objective repairs "damage" to a physical volume that occurs any time use of that volume is stopped (by a crash, or drive failure, for instance), without proper demounting as detailed in Section XIV. For instance, any volume whose use is stopped without proper demounting will contain an invalid volume map, for no attempt is made to update the volume map until demount time. Such a volume may contain an invalid VTOCE free list, as VTOCEs are freed and disk requests are executed in not necessarily the same order.

The second objective repairs damage that cannot come about simply by improper shutdown; there is no way that the storage system will allow inconsistent states to exist wherein reused addresses appear. If a reused address appears in a VTOC, it is due to undetected hardware or software failure. This is also the case if the static parameters of the volume map, for example, become inconsistent with the volume label. No accounting can be made for such damage, nor can the actual "correct" state ever be exactly determined. Such damage, which is rare, must be "corrected" to satisfy the primary goal of the physical volume salvager, the validation of storage system assumptions.

ASSUMPTIONS MADE VALID BY THE PHYSICAL VOLUME SALVAGER

The following are the assumptions about the state of a volume, which may not be true if the volume is not properly shut down, which are made true by the physical volume salvager:

1. The current-length (vtoce.csl) of each segment, in its VTOCE, describes the 1-relative page number of the highest nonnull address in the file map.

2. The records-used (vtoce.records) of each segment, in its VTOCE, is the number of nonnull addresses in its file map. Like vtoce.csl, this will not be true for active segments that suffered page creation or deletion while active and received VTOCE updates before use of the volume was interrupted.

3. The volume-map has a "0"b for every record address cited in a VTOCE on this volume, and a "1"b for every other address in the paging region.

4. The volume map has the correct number of "0"b bits in the volume map, when (3) is true.

5. Every free (vtoce.uid = "0"b) VTOCE is part of a consistent, nonlooped chain, whose head is kept in vtoc_header.first_free_vtocx. The end of the chain is -1.

6. The cell vtoc_header.n_free_vtoce describes the number of VTOCEs in the chain as described by (5).

If these assumptions are not true for a volume that is accepted, segment control, page control, and volume management will malfunction. These assumptions are always true for a volume that has been demounted properly. Thus, the acceptance of any volume that has not been properly demounted implies a volume salvage to force these assumptions true.

The physical volume salvager reports any deviance from assumptions 1 and 2. These reports may be taken as cues to the damaging of active segments by improper shutdown.

FORMS OF DAMAGE CORRECTED BY THE PHYSICAL VOLUME SALVAGER

The following further forms of damage to physical volumes are corrected, via various assumptions, by the physical volume salvager. Such damage cannot result simply from improper or non-existent shutdown. Software or hardware damage to the volume is a prerequisite.

1. An address appearing in more than one VTOCE. If one page so affected is a page of a directory and one is not, the directory is awarded the page. Otherwise, zeros (via a null address) are assigned to both pages.

2. Inconsistent maximum length (vtoce.msl less than vtoce.csl). It is set to current-length.

3. Addresses not on the legal boundaries of the paging region of the volume. They are replaced in the VTOCE file map by null addresses.

4. Inconsistency of the global volume map parameters (there were software problems creating these inconsistencies in release 4.0). They are corrected on these assumption that these known software problems (in the disk rebuilder) caused them.

OTHER VOLUME SALVAGER ACTIONS

The running of the physical volume salvager is primarily a walk through the VTOC of the physical volume being salvaged, recreating the volume map and checking individual VTOCEs. In the case where a volume that has not been properly shut down is being salvaged, and the system has an unflushed paging device, the physical volume salvager makes a call to page control (pc$flush_seg_old_pd) for each VTOCE processed, in order to repatriate pages of the segment owning the VTOCE trapped (at crash time) on the paging device. This service of page control, the post-crash PD flush, is fully described in Section IX. This service of page control is passed the file map region of the VTOCE as a parameter; page control may place disk record addresses in it, in the case where the post-crash flush resurrects addresses. The physical volume salvager's checking of current length and records-used is postponed until such resurrection has been performed.

The physical volume salvager terminates by setting the label variables label.time_map_updated and label.time_salvaged to the same value, the current time. This will cause subsequent acceptance of the volume to realize that the volume is consistent, i.e., satisfies the conditions above, and need not be salvaged again. See "Physical Volume Acceptance" in Section XIV.

THE DISK REBUILDER

The disk rebuilder is a special version of the volume salvager that copies one physical volume onto another, reassigning address and reallocating partitions. The disk rebuilder is invoked via the "rebuild_disk" operator command, described in the Multics Operators' Handbook, Order No. AM81. The disk rebuilder copies the contents of partitions, and copies VTOCEs from the source physical volume to the target. Addresses on the target volume are allocated by the rebuilder, and the contents of pages of segments copied from the target volume to the addresses so allocated via the explicit disk reading and writing mechanism described in Section XIV.

The disk rebuilder updates the VTOCEs of all active segments by searching the AST for each segment, and performing the page-control deactivation service (see Section IX) and a VTOCE update (See Section IV) for each segment found active before it is copied. This updates the VTOCE and segment pages on the disk.

The shutdown state and label times of the disk being copied are falsified by the disk rebuilder in the case where an accepted physical volume is being copied. Were this not the case, a volume being so copied would appear to have crashed during the middle of a disk rebuild.

ASSUMPTIONS NOT CHECKED BY THE VOLUME SALVAGER

The following assumptions about the storage system hierarchy must be true in order to ensure correct operation of the system. They can become invalid by interruption of operation or use of a physical volume. However, since all of these assumptions take great expenditures of real time to be made true, the system is prepared to operate without their being true. The adverse effect which will result is detailed in each case.

1. All directories have valid threads and formats. This assumptions is made valid by a full run of the hierarchy salvager. If a directory is encountered with any of various invalid threads and formats during normal operation, a crawlout will occur. The online salvager will salvage that directory, and cause this assumption to be valid.

2. Every VTOCE that is designated by a PVID-VTOC index pair in a segment or directory branch in fact is in use, and indeed is the VTOCE for that segment or directory (vtoce.uid must be the same as entry.uid). A segment or directory for which this is not true is said to suffer a connection failure (See Sections II and IV). Any primitive which accesses a VTOCE, from a branch, is prepared for this occurrence, and will return error_table_$vtoce_connection_fail. Such "segments" may be deleted, but not activated. A full run of the hierarchy salvager in "check_vtoce" mode (see the MOH) will detect and delete all such branches.

3. For every VTOCE, there must be a branch which, via a PVID-VTOC index pair, designates this VTOCE. A VTOCE for which this is not true is said to suffer reverse connection failure. The effect of this problem is wasted VTOCEs, and wasted disk records (the records designated by the file maps in such VTOCEs), as the "segments" they describe are not in any way accessible. The tool sweep_pv (see the Multics Operators' Handbook, Order No. AM81), invoked for "garbage collection and deletion", reports and deletes such "orphan" VTOCEs (See "Special Services for sweep_pv", Section IV).

4. The "quota used" cell of every directory must contain a number equal to the sum of the "records-used" fields of all immediately inferior segments, and of the "quota used" cells of all immediately inferior directories that do not have their own quota accounts. This is the definition of "quota used". When this is not so, users experience negative used figures and other false used figures, being charged for nonexistent pages or not being charged for existent pages. A full run of the hierarchy salvager in "check_vtoce" mode remedies this situation. Similarly for directory quota.

The assumptions 1, 2, and 4 are made valid by the hierarchy salvager for all directories critical to the booting of the system, during bootload, if it was determined that the system was not shut down properly (and hence the RPV, and thus the RLV, on which all directories exists) during the previous bootload. It is only in this case that these anomalies can occur. The system forces these assumptions to be true for these critical directories by automatic invocation of the hierarchy salvager during bootload and system startup.

SECTION XVI

SCENARIOS

This section gives two scenarios of typical operations in the storage system, showing who calls what and how, and what data is affected. The handling of a typical segment fault and a typical page fault are detailed in this way. These sequences are intended to be typical, not canonical.

A SEGMENT FAULT

We will consider a segment fault on >udd>x>y>z. >udd>x>y is known with a segment number of 243, and >udd>x>y>z with 244. The segment >udd>x>y>z is described by VTOCE 2045 on physical volume pub01, which is mounted on the drive whose PVT index is seven. The current length of this segment is 100K. >, >udd, and >udd>x are active, and >udd>x>y and >udd>x>y>z are not.

A reference to 244¦14 is made by the processor. A directed fault 0 occurs.

The module "fim" is invoked, recognizes this directed fault as a segment fault, and invokes the segment fault handler, seg_fault.

The segment fault handler determines that indeed there is no SDW for segment 244, and it is not a process stack.

seg_fault calls sum$getbranch_root_my with the pointer 244¦14, hoping to obtain a pointer to its branch.

sum$getbranch_root_my inspects the KST entry for segment 244, determining from kste.entryp that its branch resides (as seen by this process) at 243¦5730, in >udd>x>y. sum$getbranch_root_my calls lock$dir_lock_read to lock this directory to validate the branch.

lock$dir_lock_read tries to touch >udd>x>y, but takes a segment fault. A directed fault 0 occurs.

The module "fim" is invoked recursively, recognizes the segment fault, and invokes the segment fault handler recursively.

The segment fault handler processes the segment fault on 243¦10, performing the actions now being recursively described.

The fim is returned to, and restarts the reference made by lock$dir_lock_read.

lock$dir_lock_read places an entry in the dirlock_table, locking >udd>x>y to this process.

sum$getbranch_root_my calls validate_entryp to ensure that 243¦5730 is still the branch for z.

sum$getbranch_root_my returns the pointer 243¦5730 to seg_fault, with >udd>x>y locked to this process.

seg_fault checks the time in that branch against the time in the KSTE to ensure that the access calculated at initiate time is still valid.

seg_fault calls activate with the pointer to z's branch, 243¦5730, to receive an AST entry pointer with the AST locked.

activate copies critical information out of the branch at 243¦5730 into its stack, and locks the AST, in order to determine if >udd>x>y>z is active.

activate calls search_ast with the UID of z to search the AST for z. search_ast replies that z could not be found, and is thus not active. activate unlocks the AST.

activate calls get_pvtx with the physical volume ID of volume pub01 (from the branch at 243¦5730), to get a PVT index. This program returns "7". This number can be invalidated at any time.

activate calls vtoc_man$get_vtoce with the PVT index 7, the PVID of pub01, and the VTOC index of z's VTOCE, 2045, the latter two items culled from the branch at 243¦5730. The first vtoce-part is requested.

vtoc_man$get_vtoce locks the VTOC buffer lock, and calls GET_BUFFERS_READ to see if PVT index 7, VTOCE 2045, first vtoce-part is present. It is found by SEARCH in vtoce-part buffer 33. It is copied out to activate's stack frame, and the VTOC buffer lock is unlocked.

activate sees that z is longer than 96K, and that the second vtoce-part will be required to get the file map. Activate calls vtoc_man$get_vtoce asking for PVT index 7, VTOCE 2045, second vtoce-part.

vtoc_man$get_vtoce locks the VTOC buffer lock, and calls GET_BUFFERS_READ to find the second vtoce-part of PVT index 7, VTOCE 2045. It is not found. A vtoce-part buffer (15) is pre-empted from PVT index 6, VTOCE 1011, third vtoce-part, and the disk DIM is called to read the second vtoce-part of PVT index 7, VTOCE 2045 into it.

vtoc_man unlocks the VTOC buffer lock, having set buffer 15 out-of-service, and calls pxss$wait to wait for the event 333000000015.

The disk DIM interrupt side calls vtoc_interrupt with the main memory address of VTOC buffer 15. The out-of-service bit is turned off, and pxss$notify is called to notify the event 333000000015.

pxss$wait returns to vtoc_man, which relocks the VTOC buffer lock, and copies buffer 15 into activate's stack frame.

vtoc_man unlocks the VTOC buffer lock, and returns to activate.

activate locates the ASTE for segment 243 in this process, >udd>x>y. It is at 17¦20444.

activate calls get_aste to obtain a 256-word AST entry to hold the segment z.

get_aste inspects the first ASTE on the 256K used list. It is for >udd>m>joe>bill.list, which has 12 pages in main memory.

get_aste inspects the second ASTE on the list. It is for >udd>m>cp>temp, which has no pages in main memory, and has had none come in since get_aste last saw this ASTE. It will be deactivated.

get_aste calls deactivate, passing it 17¦24644, the address of the AST entry of >udd>m>cp>temp.

deactivate calls setfaults to destroy all SDWs for >udd>m>cp>temp.

setfaults runs down the trailer list for >udd>m>cp>temp, locating all SDWs and, accessing the descriptor segments of various processes via an SDW-level abs-seg, removes these SDWs. Setfaults calls to clear the system's associative memories.

deactivate calls pc$cleanup to get all pages out of main memory.

pc$cleanup locks the page tablelock, and finds that no pages are in main memory for >udd>m>cp>temp. It unlocks the page table lock.

deactivate calls update_vtoce to update VTOCE 2311, PVT index 6 which, as determined from aste.pvtx and aste.vtocx in the ASTE at 17|24644, are the PVT index and VTOC index of the VTOCE for >udd>m>cp>temp.

update_vtoce finds that >udd>m>cp>temp is 136K long, and no vtoce-parts will have to be read.

update_vtoce calls pc$get_file_map, passing it the AST address 17|24644, to get a copy of the AST, with definitive information.

pc$get_file_map locks the page table lock, constructs a valid copy of that ASTE in its stack, unlocks the page table lock, and copies it out to update_vtoce's stack frame. This includes the file map.

update_vtoce constructs an image of VTOCE 2311 on PVT index 6, first two vtoce-parts, from the information returned by pc$get_file_map.

update_vtoce calls vtoc_man$put_vtoce with a zero as PVID, the PVT index 6, the VTOC index 2311, the image of the first two vtoce-parts of this VTOCE, and a request to write out these vtoce-parts.

vtoc_man$put_vtoce locks the VTOC buffer lock, and searches for buffers containing these vtoce-parts. None do. GET_BUFFERS_WRITE causes two other vtoce-parts to be preempted, and returns to vtoc_man$put_vtoce their indices, 23 and 16.

vtoc_man$put_vtoce copies the two vtoce-parts supplied by update_vtoce into buffers 23 and 16 respectively, setting these buffers out-of-service.

vtoc_man$put_vtoce calls the disk DIM to start writing the two buffers 23 and 16, and unlocks the VTOC buffer lock.

vtoc_man$put_vtoce returns to update_vtoce with the I/O still in progress.

update_vtoce determines that no nulled addresses were culled by pc$get_file_map, and VTOCE I/O completion will not have to be awaited.

update_vtoce returns to deactivate, having updated VTOCE 2311 on PVT index 6.

deactivate calls put_aste to free the ASTE at 17|24644. It is moved to the head of the used list.

get_aste returns the ASTE at 17|24644, now free, to activate.

activate connects the ASTE for >udd>x>y (17|20444) with the ASTE to be used for >udd>x>y>z at 17|24644.

activate calls pc$fill_page_table to fill in the ASTE's page table with information in the VTOCE (2045) on PVT index 7 which has been read in.

pc$fill_page_table converts the formats of the device addresses, and initializes the PTWs in this ASTE. A check is made for reused addresses.

The disk DIM interrupt side calls vtoc_interrupt, placing VTOC buffer 23 no longer out-of-service.

activate fills in the activation attributes of >udd>x>y>z into the ASTE at 17¦24644, along with other information.

activate returns to seg_fault with the AST locked, returning the ASTE pointer 17¦24644 for >udd>x>y>z.

seg_fault sets the encacheability state of >udd>x>y>z to "one process, reading and writing, encacheable".

seg_fault constructs an SDW for >udd>x>y>z, and places it at slot no. 244 in this process' descriptor segment.

seg_fault constructs a trailer entry in str_seg for this descriptor, giving the number 244 and the ASTE offset of this process' descriptor segment.

seg_fault unlocks the AST.

The disk DIM interrupt side calls vtoc_interrupt, placing VTOC buffer 16 no longer out-of-service.

seg_fault calls lock$dir_unlock to unlock >udd>x>y.

seg_fault returns to the fim.

The fim restarts the machine conditions for the segment fault.

The process proceeds, and the segment fault has been resolved.

A PAGE FAULT, IN PAGE MULTILEVEL

Having resolved a segment fault on >udd>x>y>z, our user process next attempts to access location 14.

The appending unit finds PTW 0 for segment 244 (at 17¦14660) to have ptw.df off. A directed fault 1 occurs.

The page fault handler, page_fault, is invoked. It saves all registers and machine state at pds$page_fault_data. It sets up a stack frame on the PRDS.

page_fault attempts to lock the page table lock, but finds it locked.

page_fault branches to pxss$ptl_wait to wait for the page table lock.

pxss$ptl_wait locks the traffic controller lock. It finds that the page table lock is indeed still locked. The cell sst.ptl_wait_ct is incremented.

This process is made to wait for the "PTL Event".

The process which had been holding the page table lock unlocks it, but notices sst.ptl_wait_ct nonzero.

The process which had been holding the page tablelock calls pxss$page_notify to notify the "PTL Event".

Our process resumes. pxss$ptl_wait returns to page_fault$wait_return, and the page fault is restarted.

The appending unit finds PTW 0 for segment 244 (at 17¦24660) to have ptw.df off, all over again. A directed fault 1 occurs.

The page fault handler, page_fault, is invoked. It saves all registers and machine state at pds$page_fault_data. It sets up a stack frame on the PRDS.

page_fault attempts to lock the page table lock, and succeeds.

page_fault calls pd_util$check_pd_free_and_update to see if the paging device needs housekeeping.

pd_util$check_pd_free_and_update determines that the PDMAP has been written out in the last second, and will not write it out.

pd_util$check_pd_free_and_update sees that only 8 PD records are free. 10 must be free or being freed.

pd_util$check_pd_free_and_update walks down the PD used list to find entries to free. Eight PD records are skipped, and the one at 17¦6440 is found. It is found to describe a PTW at 17¦15262, which describes a page in main memory. It is not a good candidate for replacement.

The next PD record on the used list, at 17¦6224, similarly describes a page in main memory, and is skipped.

The next PD record on the used list, at 17¦6030, describes a page not in main memory. It is not "PD Mod" (pdme.mod = "0"b).

pd_util$check_pd_free_and_update evicts this page from the paging device, taking the disk device address at 17¦6031 and placing it in the PTW (17¦17327) pointed at by pdme.ptwp at 17¦6032.

pd_util$check_pd_free_and_update calls pd_delete_ in the same program to put the PDME at 17¦6030 in the used list as free. The count of free PDMEs is now 9.

pd_util$check_pd_free_and_update considers the next PDME in the list, the one at 17¦6204. It describes a page whose PTW is at 17¦22137, and a nulled disk address (401512) on PVT index 6.

pd_util$check_pd_free_and_update calls rws_ in the same program, to start an RWS for the PD record whose PDME is at 17¦6204.

rws_ calls page_fault$find_core to find a main memory frame in which to perform the RWS.

page_fault$find_core picks up sst.usedp, which has the value 1550.

The core map entry at 17¦1550 is inspected. It describes a page whose PTW at 17¦21532 is modified with respect to paging device or disk (ptw.phm = "1"b). This page is not acceptable for eviction.

The core map entry at 17¦1304 is pointed at by cme.fp of the one at 17¦1550. It describes a page whose PTW at 17¦16120 describes a pure page which was recently used. This page is not acceptable for eviction.

The core map entry at 17¦1340 is pointed at by cme.fp of the one at 17¦1304. It describes a page whose PTW at 17¦17172 indicates that this page is pure, and not "not-yet on paging device".

The access to the page whose PTW is at 17¦17172 is turned off, by turning off the directed fault bit in that PTW, and clearing the system's associative memories, and clearing the caches of that page's words.

find_core calls page_fault$cleanup_page, which puts the PD address in the CME at 17¦1340 back in the PTW at 17¦17172, and adjusts the AST entry for this segment at 17¦17154. The CME at 17¦1340 is freed.

find_core returns the CME at 17¦1340 to rws_.

rws_ threads out the PDME at 17¦6204 and the CME at 17¦1340, and cross-relates them to indicate the read cycle of an RWS.

rws_ calls the bulk store DIM to read PD record 41 (whose PDME is at 17¦6204) into location 700000 in main memory whose CME is at 17¦1340. The bulk store DIM starts this read.

rws_ returns to pd_util$check_pd_free_and_update, who now notes that there are 10 PD records free or being freed.

pd_util$check_pd_free_and_update notes that there are incomplete RWS reads, and calls the bulk store DIM in a loop until there are none.

At one of these times, the bulk store DIM notices status for the read into location 700000, and calls page_fault$done_ with the address 700000, and an error code of zero.

page_fault$done_ locates the CME at 17¦1340, and inspects it, noting that an RWS read was in progress. The routine read_write_sequence in done_ is invoked.

page_fault$done_ acknowledges the RWS read completion, and indicates in the CME at 17¦1340 that an RWS write is in progress. The disk DIM is called (via device_control$dev_write) to write the address 001512 on PVT index 6 from location 700000.

page_fault$done_ returns to the bulk store DIM.

The bulk store DIM returns to pd_util$check_pd_free_and_update.

pd_util$check_pd_free_and_update notices that there are no more RWS reads outstanding, and returns to the page fault handler.

The page fault handler inspects the SCU data at pds$page_fault_data, and determines that this is not a descriptor segment page fault.

The page fault handler locates the SDW for segment 244, implicated in the machine conditions, and subsequently its page table at 17¦24660 and its ASTE at 17¦24644.

The page fault handler inspects the PTW at 17¦24660, and determines that indeed a page fault situation exists.

The page fault handler calls read_page, passing it the PTW address 24660 (relative to the SST), requesting the allocation of a main memory frame and subsequent readin of a page.

read_page checks that a nonnull address exists in the PTW at 17¦24660. It is address 002167 on PVT index 5, which is, by virtue of its format, nonnull and nonnulled (live). Thus, no quota check or allocation will be necessary.

read_page calls find_core to get a main memory frame into which to read that address.

find_core inspects the first CME on the main memory used list. This CME, at 17¦2200, was pointed to by cme.fp of the one at 17¦1340, which is now threaded out as an RWS is in progress there.

The CME at 17¦2220 describes a page whose PTW (at 17¦14140) indicates ptw.nypd, requiring allocation to the paging device. This page is not suitable for eviction. Its cme.fp pointer designates the CME at 17¦2214.

The CME at 17¦2214 designates a page that is neither modified nor "not yet on the paging device". Its PTW is at 17¦15150, and it will be evicted.

The access to the page whose PTW is at 17¦15150 is turned off, by turning off the directed fault bit in that PTW, and clearing the system's associative memories, and the caches of that page.

find_core calls page_fault$cleanup_page, which puts the PD address in the CME at 17¦2214 back in the PTW at 17¦15150. The AST entry at 17¦15130 is adjusted appropriately, and the CME at 17¦2214 is freed.

find_core returns the CME at 17¦2214 to read_page.

read_page sets the CME at 17¦2214 to indicate a page out-of-service on a read. The disk address 002167 is copied from the PTW at 17¦24660 to this CME. It is threaded out of the core used list. The main memory address, 230000 is placed in the PTW at 17¦24660, but ptw.df is still off.

The disk DIM is invoked, via device_control$dev_read, to read the record 002167 from the disk on PVT index 5 into location 230000 in main memory, the location described by the CME at 17¦2214. The read is started.

read_page returns to the page_fault_handler, informing it that waiting will be necessary.

The page fault handler calls claim_mod_core to start all I/Os that were skipped by find_core during this page fault.

claim_mod_core inspects sst.wusedp, which describes the CME at 17¦1550. This page was skipped because it needed writing.

claim_mod_core calls write_page, passing it as an argument the CME at 17¦1550.

write_page checks this page for zeros. It does not contain zeros.

write_page calls allocate_pd to see if this page requires allocation to the paging device.

allocate_pd notices that this page is already on the paging device (at record 101), and returns this fact to write_page.

write_page threads the CME at 17¦1550 out of the used list, marks the PTW (at 17¦21532) out-of-service, and marks the CME out-of-service on a write.

write_page calls the bulk store DIM to write the main memory frame at location 264000 (described by the CME at 17¦1550) to record 101 of the bulk store.

write_page returns to claim_mod_core.

claim_mod_core inspects the next CME that was in the used list, at 17¦1304. This CME was skipped because its PTW at 17¦16120 described a recently used page. The bit indicating this (ptw.phu) in this PTW is turned off, as demanded by the main memory replacement algorithm.

claim_mod_core inspects the next CME that was in the used list, the one at 17¦2200 was skipped because it designated a page which required migration to the paging device.

claim_mod_core calls write_page, passing the CME at 17¦2200 as an argument.

write_page notices that this page is pure, and does not check for zeros.

write_page calls allocate_pd to see if this page needs allocation to the paging device.

allocate_pd sees that the bit ptw.nypd is on in the PTW at 17¦14140, and determines that this page must be allocated a record of paging device.

allocate_pd inspects the first PDME on the PD used list. It is at 17¦6044, describing record 11 of the bulk store, and is free.

allocate_pd moves the PDME at 17¦6044 to the tail of the PD used list, and fills it with information from the CME at 17¦2200. The CME at 17¦2200 is changed to designate record 11 of the paging device as the home for the page.

allocate_pd returns to write_page the fact that a PD record was allocated during this call.

write_page sets the CME at 17¦2200 out-of-service, threading it out of the used list, and marks the PTW at 17¦14140 out-of-service.

write_page calls the bulk store DIM to write the main memory frame at 240000 (described by the CME at 17¦2200) to record 11 of the bulk store.

write_page returns to claim_mod_core.

claim_mod_core notices that sst.usedp and sst.wusedp are equal, and all operations skipped by find core have been processed.

claim_mod_core returns to the page fault handler.

The page fault handler determines that the PTW for the page faulted on (page 0 of segment 244) is still marked out-of-service (at 17¦24660). Since it is not a page on the paging device, the traffic controller will be used for waiting.

The page fault handler meters the page fault and the time spent in processing it.

The page fault handler develops the event ID for the page faulted on, 024660, and stores it in pds$arg_1 for pxss. The bit cme.notify_requested is set in the CME at 17¦2214.

The page fault handler branches to pxss$page_wait, with the page table locked.

pxss$page_wait locks the traffic controller lock.

pxss$page_wait unlocks the page table lock.

pxss$page_wait sets the process to waiting on the event 024660, and uses the PRDS frame to switch processes to another process.

Our process goes waiting.

The I/O operation in location 700000 completes, and the disk DIM interrupt side calls page$done with this number as a parameter.

page$done locks the page table lock and calls page_fault$done_ with that main memory address.

page_fault$done_ locates the core map entry at 17¦1340, and seeing that an RWS was in progress there, calls the routine read_write_sequence in done_. The PDME at 17¦6204 is located.

page_fault$done_ notices that a write cycle completed. The disk address 401512 (nulled) is resurrected to 001512, and taken from the PDME at 17¦6204 and placed in the PTW at 17¦15263, which had contained the paging device address 000041.

page_fault$done_ frees the PDME at 17¦6204, and frees the main memory at location 700000, placing the CME at 17¦1340 into the core used list.

page_fault$done_ returns to page$done, which unlocks the page table lock.

page$done returns to the disk DIM.

The I/O in location 230000 completes. The disk DIM interrupt side calls page$done.

page$done locks the page table lock and calls page_fault$done_.

page_fault$done_ finds the CME at 17¦2214, and sees that no RWS was going on there.

page_fault$done_ marks the CME at 17¦2214 as no longer out-of-service, threading it back into the used list.

page_fault$done_ locates the PTW at 17¦24660 from the CME at 17¦2214. The directed-fault bit is turned on, allowing access to the page, and the out-of-service bit turned off.

page_fault$done_ notices the bit cme.notify_requested in the CME at 17¦2214, and calls pxss$page_notify with the event 024660.

pxss$page_notify locks the traffic controller lock, and notifies our process, which was waiting on event 024660, and sends a pre-empt connect to CPU B.

pxss$notify unlocks the traffic controller lock, and returns to page_fault$done_.

page_fault$done_ returns to page_done, which unlocks the page table lock.

page$done returns to the disk DIM.

CPU B takes a pre-empt connect, and calls pxss$pre_empt.

pxss$pre_empt locks the traffic controller lock, and performs a "getwork" operation, abandoning the process that took the connect.

pxss (getwork) finds our process ready, and switches to it, setting it as running.

pxss (getwork) returns to pxss$page_wait in our process.

pxss$page_wait unlocks the traffic controller lock, abandons its PRDS stack frame, and transfers to page_fault$wait_return.

page_fault$wait_return restarts the machine conditions (at pds$page_fault_data) indicating an Appending Unit address preparation fault.

CPU B's Appending Unit successfully fetches and uses the PTW at 17¦24660, and resolves the virtual address 244¦14 to absolute address 230014.

SECTION XVII

GLOSSARY

abort

See RWS abort.

abs-seg

Not a segment at all. A segment number used for addressing as a segment an arbitrary main memory, disk, or paging device extent, the location of the extent being a parameter at the time that it must be addressed. See PTW-level abs-seg and SDW-level abs-seg.

abs-usable

A main memory frame, part of the paging pool, in a system controller that cannot be deconfigured. Only abs-usable page frames can contain abs-wired pages.

abs-wire

Of a page or segment. To make that page or segment abs-wired.

abs-wired

1. Of a page. A page in main memory, which not only is wired, but may not be moved around main memory. Pages wired but not abs-wired may be moved around by abs-wiring pages or deconfiguring memory. Used principally for I/O buffers.

2. Of a segment. A segment having some or all of its pages abs-wired.

accept

Of a physical volume. To make those supervisor calls, which, by placing label information in the PVTE for a given drive, establish the binding, or association, between that drive and that volume.

access control segment (ACS)

A segment whose ACL effectively determines access to a resource. ACSs for peripheral devices are in >sc1>rcp, and are maintained and used by RCP. ACSs for logical volumes are usually, but need not be, in >lv.

activate

To make a segment active. Done by getting an ASTE, reading the VTOCE of the segment, filling in the ASTE, and hashing it into the AST hash table. See active. The parent directory of a segment must be locked in order to activate it.

activation attributes
or activation information

Those attributes of a segment that are read from the VTOCE every time a segment is activated, copied into the ASTE, changed while the segment is active, and updated back to the VTOCE. Examples: current length, maximum length, date-time modified. See permanent information.

active

1. Of a segment. Having a page table (and AST entry) in main memory; the criterion for whether or not a segment is active is whether or not it is hashed into the AST hash table.

2. (loosely) of a page. Belonging to an active segment.

3. Of the paging device, or an instance thereof. In use, having pages being allocated, read and written from it. The bit fsdct.pd_active tells whether or not the paging device is active. See unflushed.

add_type

A subfile of page control device addresses (devadds) that specifies whether it is a record of disk, a record of PD, a main memory frame, or a null address.

append

(verb) To combine an address (effective address) produced by the processor control unit, with a segment number (the effective segment number) maintained by the appending unit, and produce, by fetching and inspecting PTWs and SDWs, either a main memory address or a page or segment fault.

appending unit (APU)

That portion of the 68/80 processor responsible for the implementation of segmentation and paging. It performs appending, maintains all segment numbers, performs control operations on its data, and coordinates the taking of faults.

appending unit cycle

One of the operations of the appending unit that results in an address being presented to the processor port logic. For an append that does not result in a fault, the last such address (final address) is the address of the data requested by the control unit. Other APU cycles are to obtain and modify PTWs or SDWs. Each CU cycle (see control unit cycle) may produce many APU cycles.

associative memory

A content-addressable semiconductor memory in the processor appending unit. There are two: the PTW associative memory, which maintains the last 16 PTWs fetched from main memory, and similarly the SDW associative memory. The associative memories can be cleared via the CAMS and CAMP instructions. Used for speed, to avoid continual PTW and SDW fetching from main memory.

AST

For active segment table. The collection of ASTEs that describe all of the active segments in the system. (See ASTE.) The AST is the uppermost part of the SST.

ASTE

For active segment table entry.

1. (ASTE proper). A collection of attributes, other than file map or page tables, describing an active segment.

2. That collection of attributes, taken along with the page table and file map of a segment.

AST hash table

A table, kept in active_sup_linkage, that holds the heads of hash threads, so that the UID of any segment may be used to find its ASTE, if it is active, or the fact that it is not active.

AST pool

There are four sizes of AST entries, those containing page tables of 4, 16, 64, and 256 PTWs. Those of each size form four <u>pools</u>, that are managed separately.

AST trickle

A mechanism implemented in the AST replacement algorithm that periodically updates, from the AST, the VTOCEs of segments whose VTOCEs would benefit by so being updated. It is driven by AST traffic.

AST used lists

One of seven lists of AST entries, the four normal lists being "used lists" of each size, and others being lists of ASTEs selected by some special criteria. See used list.

atomic

See unitary.

attached

1. A private logical volume is attached to a process if an entry in that process' KST so attests. Only private logical volumes that are attached to a given process can be used by that process.

2. There are definitions of this term relative to user-ring I/O and resource control.

auxiliary service

A service provided by a subsystem that, although it is not one of the fundamental ones for which the subsystem exists, involves much of the central code of the subsystem. See peripheral service and basic service.

bad track list

A reserved area of the volume header of a physical volume, that will be used to contain bad track information in future releases.

basic service

A service provided by a subsystem that is one of the fundamental ones for which it exists. See auxiliary service and peripheral service.

bit map

A one-bit-per-record map of all of the disk records usable for paging on a given physical volume. All bit maps (those for mounted physical volumes) reside in the FSDCT.

bootload

1. (verb) To initiate the operation of the Multics system, when it is down, i.e., to bring it up via issuing the BOS BOOT command.

2. (noun) The act of bootloading.

3. The life-span of a Multics hierarchy from time of bootload to shutdown, or next bootload, in the case where shutdown cannot be performed.

branch

As used in this document, a data structure in a directory that describes a segment or directory. A segment's branch contains a physical volume ID and VTOC index for the VTOCE of a segment or directory. The ACL, names, author, bit count, etc., of a segment may be found in or from its branch.

bulk store

A core memory storage medium used as a paging device. The term is used when it is not relevant that it is being used as a paging device as opposed to any other storage medium.

cache

A 2048-word semiconductor buffer memory in the processor port logic. An attempt is made to maintain the last 2048 words fetched from main memory in the cache. The cache provides a substantially faster access time than that of main memory. As each processor contains its own cache, strategies are needed to prevent confusion about main memory contents. See encacheable.

call side
Those programs in page control that are explicitly invoked by processes that need services performed upon segments. See fault side and interrupt side.

call-side wait coordinator
The program (page$pwait) used by call-side page control to wait for a given event, via an appropriate mechanism, and set whatever bits will be necessary to cause the occurrence of the event to perform the necessary notification.

claim
When the PD or main memory replacement algorithm selects a page frame to have its contents evicted, and thus be freed, the page frame is said to have been claimed.

connected
Of a process and a segment. A segment is said to be connected to a process, or vice versa, if the descriptor segment of that process contains an SDW that describes that segment, and is not faulted. See trailer.

connection failure
or forward connection failure
A situation that exists when a directory branch describes a certain VTOCE, but that VTOCE describes some other (or no) segment. This can be determined by comparing UIDs. This situation can come about by accident or by deliberate salvager action. See also reverse connection failure.

contract
The action performed by a program, the conditions that must be true when it is invoked, and the circumstances describing the meaning and validity of the result.

control unit (CU)
That portion of the 68/80 processor that is responsible for decoding instructions, performing indirections, and routing data around the processor. The control unit develops effective addresses of words in segments; see appending unit. The control unit status can be stored via the SCU instruction, when a control unit cycle is aborted due to a page or segment fault.

control unit (CU) cycle
A control unit operation resulting in an effective address being presented to the appending unit or port logic. Typical CU cycles are "instruction pair fetch," "operand fetch," "indirect word fetch," etc. Any CU cycle can result in a page or segment fault when appending is performed for that cycle. Restart of that fault retries the aborted CU cycle.

core
An obsolete term used in many program listings and comments for main memory.

core map
A page control data base in the SST that contains a four-word entry (CME) for frame of main memory. It is protected by the page table lock. See Section VI for its layout. See core.

core used list
or main memory used list
A used list of core map entries (CMEs) describing the order of recency of use of main memory frames.

crawlout
A cross-ring signal that causes abandonment of a stack in the inner ring. Crawlouts out of ring 0 require supervisor data bases to be cleaned up. Crawlouts result from hardware problems or faulty software, or damaged directories, and cause the software running at the time to be interrupted and not continued.

critical process pages

The first page of a process' descriptor segment and PDS. These two pages must be wired (the process is then loaded) before the process can actually run.

deactivate

Of a segment. To remove it from the state of being active. To deactivate a segment, its pages are driven out of main memory and paging device, its VTOCE updated from its AST entry, and the ASTE freed. See active.

deadlock
or deadly embrace

A situation wherein a process having a given resource is waiting for some other process to free a second resource, but, unfortunately, the process having the second resource is waiting on the first process to free that first resource. See locking hierarchy.

deciduous

Of a segment. A segment read in as part of the bootload tape in collection 1 or 2 (i.e., part of initialization's, therefore the initializer's, hardcore address space) and placed into the hierarchy by the program init_branches. Deciduous segments reside entirely in the hardcore partition. See also reverse-deciduous.

Examples: >sl1>pl1_operators_, >pdd>!zzzzzzbBBBBBBB>pds

defined

Of a logical volume. Either a mounted public logical volume, or a mounted private logical volume attached to a given process. A process is said to have a given logical volume either defined or not defined.

demount

1. Of a physical volume. To dissociate a physical volume from the drive on which it is mounted, stopping use of it by processes.

2. Of a logical volume. To remove the logical volume table entry for a logical volume.

demounter

The procedure demount_pv, that coordinates the demounting of physical volumes.

deposit

To deposit an address to the free pool of records of a given physical volume (bit map) is to mark it as free, and available for subsequent withdrawing. See withdraw and bit map.

descriptor segment (sometimes DSEG)

An array of hardware control words (SDWs) that specifies the mapping between segment numbers and either segments or taking a segment fault. Each process has its own descriptor segment; it is a segment, and may be paged, in which case it is described by the descriptor segment page table. The first page of the descriptor segment of a loaded process is wired.

desperation

Action taken by paging device allocator when there are no free PD records. Desperation consists of evicting some nonmodified page from the PD, if it can be found near the head of the PD used list.

devadd (device address)

A page control format for all main memory, paging device, and disk addresses. The upper 18 bits are a record number or address, and the lower four bits (add_type) specify whether it is main memory, disk, paging device, or null.

DIM

For device interface module. The program that contains the code for managing the physical operation (as opposed to logical use) of a device (viz., the disks or bulk store).

double-write

A disk write performed as a reliability feature after a successful paging device write. Double writing is controlled by a parameter on the DEBG CONFIG card, and tends to keep paging device pages pure.

eligible

A process is made eligible by the traffic controller at the time that the latter describes that the former should be allowed to consume main memory resources (i.e., take page faults). Only eligible processes can run, although they must be loaded first.

emergency shutdown (ESD)

A set of procedures, invoked via the BOS ESD command, that attempt to produce an orderly shutdown of the system after a crash has occurred. This shutdown must be performed in order to update the disk records and VTOCEs for segments that were active at the time of the crash.

encacheable

Said of a segment. A segment is encacheable if words of that segment are allowed to be put (and hence subsequently found) in a processor's cache. An SDW bit (sdw.cache) controls encacheability. This is used to control sharing and prevent confusion about cache contents. Segments accessible to the IOM or FNP are routinely nonencacheable.

entry

(loosely) same as branch.

evict

Of a page. To drive a page out of a given main memory frame, or PD record, by writing it, moving it, or simply changing the state of its data bases as appropriate. Note that the eviction of pages that are identical to copies on disk or PD does not involve writing.

exposed

A physical volume is exposed to an instance of a paging device if that instance is active while that volume is mounted.

fatal

Of a crash. A crash for which a successful emergency shutdown could not be attained. Fatal crashes involve salvaging all physical volumes mounted at the time of the crash.

fault side

Those programs in page control that are invoked in response to a page fault. See call side and interrupt side.

fault vector

A pair of instructions at a fixed location in main memory associated with a specific type of fault condition. When the processor recognizes such a condition, it "takes" the fault by executing these instructions. In Multics, they are always SCU (store control unit) and TRA (unconditional transfer).

file map

A mapping between the pages of a segment and disk record addresses on some physical volume. Each page is mapped into either one such address, or a null address, indicating zero contents. File maps appear in VTOCEs. When a segment is active, the file map is distributed between the various page control data bases.

file-system time

A 36-bit representation of real clock time used in directories and in recording date-time used, date-time-modified, and other storage system times. It is the upper 36 bits of a 52-bit clock time.

frame

or "main memory frame"

A 1024-word block (on a 1024-word boundary) of main memory. See page.

FSDCT

For file system device configuration table. A paged data base containing many global volume management parameters, and all bit maps.

fsmap

Either the bit map of the volume map of a physical volume, or the bit map in the FSDCT for that volume, when it is accepted.

fsmap tail

in the fsamp of a paging region that is not a multiple of 32_{10} pages, the bits of the last word of the fsmap that are not part of the valid portion of the bit map. They must be zero.

function

A body of code that performs some particular action as part of the operation of some subsystem. A function is used in this book to mean an important internal interface. See service for comparison.

half_lock
or read lock

A data object that allows many processes to perform certain actions upon a data object or data base, but does not let certain other actions begin until no processes are performing any actions at all (either first or second kind). A process performing the second kind of action causes any process attempting to do the first kind to wait. Also called "multiple-reader one-writer" lock.

hardcore partition

A partition used for holding the pages of the supervisor. It is always on the root physical volume (RPV).

hardcore segment

A segment addressable in the hardcore address space of all processes. All such segments are created via initialization.

hierarchy

A set of directories, segments, and volumes that describe each other completely. Normally each site maintains one hierarchy, although some maintain others for development use. All unique IDs and PVT indices, paging device information, etc., are valid only with respect to one hierarchy.

higher

Of locks. See locking hierarchy.

"hot" buffer

A vtoce-part buffer, that although not out-of-service, is known to have contents differing from disk. Hot vtoce-part buffers arise only as a result of write errors, and must be flushed at demount time.

inhibit

Of a physical volume. To prevent segment (and VTOCE) creation upon that volume, by the setting of a bit (pvte.vacating) in its PVTE. This can be done via the inhibit_pv command, or the sweep_pv command.

initialization

The set of programs that run when Multics is bootloaded, until it is up to command level. Initialization is responsible for creating the supervisor data bases, and building the hardcore address space of all processes, among other tasks.

instance

Of the paging device. The paging device, and all of the pages that are and have been on it, and its map, from the time its map is initialized to the time it is shut down or the last record is flushed from it, whichever happens first.

interrupt side

Those programs in page control that are called by storage system DIMs to respond to the completion of an I/O operation. The interrupt side is not only invoked on behalf of interrupts; "running" and other activity can invoke it as well.

kernel

Of page control. The ALM programs in the main path of the page fault handler.

known segment table (KST)

A per-process table describing the mapping between segment numbers in that process and storage system segments. The segments are identified via pointers to their branches (using other segment numbers in that process) and unique IDs. The KST also contains a list of private logical volumes attached to the process. The KST is a reverse-deciduous segment.

label
or volume label

The first Multics record of a physical volume. It identifies the volume, and gives parameters about its last use.

live

Said of a disk address. A disk address that represents a given record of disk and the data in it. See nulled for comparison.

loaded

A process is loaded when its two critical process pages have been wired. Processes are loaded by the traffic controller when they are made eligible. Only loaded processes can actually run.

lock

A data object used to serialize processes performing certain actions and using or modifying certain data bases. A process locks a lock before performing these actions or using these data bases, and unlocks it when done. Only one process may have a lock locked at one time. A process trying to lock a lock that is locked by (or to) another process must wait for that lock. Processes are also said to hold locks when they have them locked.

In Multics, locks are single words of storage that are zero when not locked and contain the process ID of the process that has it locked when it is locked. See protect.

locking hierarchy

A conceptual partial-ordering of a set of locks via the arbitrary relation "higher" (>). If lock A >lock B, and lock B >lock C, then lock A >lock C. There is no inverse, and two locks may be totally unrelated. The locking hierarchy is used to prevent deadlock. The rule used by the Multics supervisor, states that no process may wait for the unlocking of a lock unless that lock is higher than every lock it has locked (sometimes called the "Bensoussan Algorithm").

logical volume

A set of physical volumes defined as a group, to which the user may direct the creation of segments. The choice of physical volumes with a logical volume for a creation is a dynamic choice of the system, and segments may move automatically between physical volumes of a logical volume.

logical volume table (LVT)

A system data base (in the segment "lvt") that contains per-logical volume parameters for each mounted logical volume, as needed by the hardcore. Included in such information is access class, and the heads of a thread of PVTEs of physical volumes belonging to that logical volume.

LVTE

For logical volume table entry. See logical volume table (LVT).

machine conditions
A 40-word description of a processor state at the time of a fault or interrupt. It contains the contents of all program-accessible registers, the state of an aborted control unit cycle, and various other information. To restart a set of machine conditions is to cause the processor to load its registers from that machine state, and resume the interrupted program.

main memory
(formerly core)
The core or MOS memory device from which the processor normally fetches instructions and data. All pages must be in main memory to be directly used by the processor. See also core.

master directory
A directory whose quota is not derived from its parent, and cannot be returned to it. Master directories are the only directories whose sons logical volume can be different from that of the parent directory. The setting of quota accounts on master directories, and the creation and deletion of master directories, is controlled by master directory control in ring 1.

migrate
To migrate a page to the paging device is to allocate a PD record for it, and write it to that record. From then on, it will be read from that record, until it is migrated off it.

mounted
1. Of a physical volume. Being physically mounted, and having the PVT entry for the drive on which it is mounted filled with parameters of that pack (pvte.used will be on).

2. Of a logical volume. Having all of its physical volumes mounted (1), and having an entry in the LVT.

multiplex wait protocol
The technique used by call-side page control to wait for a large number of events in parallel; it involves the simplex wait protocol and waiting for an arbitrary event.

multistep operation
An operation consisting of many unitary operations. See unitary operation.

nondeactivateable activation
Same as semipermanent activation.

nondeciduous hardcore segment
A paged hardcore segment that is not deciduous. Such segments are not in the storage system hierarchy, and thus have no pathnames.

Examples: bound_file_system, bound_system_faults.

not-yet-on-paging-device (nypd)
A pure page in main memory that has not yet been migrated to the paging device. Such pages are important because their eviction requires migrating them to the paging device.

null
An address that represents a record of any device, and a logical page content of zeros. A null address in a VTOCE is represented by having its high-order bit on. In page control, it is represented by an "add_type" of zero. The low-order 17 bits of a null address contain a debugging code that reflects the manner in which it was generated.

nulled
Of a disk address. One that represents a given record of disk, but a logical content of zeros. Nulled addresses appear only in page control data bases, never in VTOCEs. They may not be reported to VTOCE file maps. See null and live.

oopv

For out of physical volume. A condition where no more free records exist on a physical volume.

orphan

(Of a segment or its VTOCE.) A segment that has a VTOCE but no branch in the storage system hierarchy. Orphans may result via certain actions of the salvager or certain crashes. They can be located via the sweep_pv command. See reverse connection failure.

out of service

Undergoing I/O. Does not imply inaccessible.

pack

A demountable unit of disk storage. Same as physical volume.

page

A 1024-word extent of data at a 1024-word boundary of some segment. Pages belong to segments; they can exist in main memory frames, or on disk records or PD records or any combination of those.

page fault

An exception condition detected by the processor hardware (the appending unit) when an attempt is made to use a PTW that specifies that some page of some segment is not in main memory. This is indicated by the bit ptw.df being off. This causes the unconditional execution of a specific fault vector that effects a transfer to the page fault handler.

paging region

That extent of a physical volume described by the volume map, in which all records described in VTOCEs reside.

page table

The array of PTWs that specifies the mapping between addresses in a segment and either main memory frames or page faults. The page table of a segment is part of the ASTE; only active segments have page tables. The SDW of a segment (a paged segment) contains the absolute address of its page table.

paging device (PD)

An optional storage device from which pages are read and written from main memory, on which copies of disk pages are maintained for faster access. Only a bulk store subsystem can currently be used as a paging device. See bulk store.

paging device map
or PD map
or PDMAP

A page control data base in the SST that contains a four-word entry (PDME) for each record of paging device. It is protected by the page table lock. See Section VI for its layout.

parasite

A segment residing on a physical volume that has no VTOCE (e.g., descriptor segments on the RPV). All such segments are currently on the RPV. Their existence implies the need for a "short RPVS" in the case of a crash.

partition

A region (extent) of a physical volume, other than the VTOC and label area, used for some other purpose than pages of storage system segments.

PD

See paging device.

PD flush

The software that runs in subsequent bootloads after a fatal crash which repatriates pages on an unflushed paging device to their disk records.

PDS

For process data segment. A per-process (reverse-deciduous) hardcore segment that contains all per-process information needed by a process other than that describing its segment number to segment mapping. The first page of the PDS of a loaded process is wired. See KST.

PD used list

A used list of paging device map entries (PDMEs), describing the order of recency of use of paging device records.

peripheral service

A service of a subsystem highly removed from the main path and procedures, that may only call very high-level interfaces of that subsystem. See basic service and auxiliary service.

permanent attributes
or permanent information

Those attributes of a segment, stored in the third vtoce-part of its VTOCE, that are rarely read or changed. Examples: UID pathname, date-time VTOCE created. See activation information.

perm wired

1. Permanently (i.e., since bootload) wired. See wired.

2. Sometimes used to mean unpaged, since such segments, indeed, cannot be removed from main memory in any way. See "temp wired."

physical volume

Same as pack. A (usually) detachable storage medium of disk storage, containing entire segments, and a VTOC containing VTOCEs describing these segments. Each segment resides wholly on one physical volume. Each physical volume belongs to one and only one logical volume.

physical volume table (PVT)

A wired table of entries (PVTEs) describing almost all of the per-pack parameters for mounted packs. There is information here for page control, segment control, and volume management. The bit map for the pack, and per-logical-volume information is not stored here.

port logic

That portion of the 68/80 processor that selects system controllers, transfers commands and addresses to them, and receives data and notification from them. The port logic receives data from the processor, but addresses only from the appending unit. It contains the cache as well.

post

To post an I/O operation that was initiated by page control is to perform those actions taken by the interrupt side when told of the completion of this action by the appropriate DIM. Posting operations may or may not involve notification.

post-crash PD flush

See PD flush.

private

A logical volume is private if it was registered with this attribute. A private logical volume must be explicitly attached by any process that wishes to use segments on it; this is done conditionally depending upon the ACS of that volume.

preacceptance

The actions taken in initialization to use the partitions, including the hardcore partition, on the RPV, before the RPV has been accepted. Preacceptance of the RPV is performed by init_pvt.

prewithdraw

To assign a disk record address to a page of a segment at the time the segment is created, or at a given explicit time, as opposed to time of first use. All supervisor segments have their pages prewithdrawn by make_sdw.

preseek

An action taken by the main memory replacement algorithm to find a page frame to claim. It "looks ahead" for a usable page, postponing writing (see "writebehind") for later.

private

Of a logical volume. One to which access by users is restricted to those specified by the ACL of the ACS for that volume. Users wishing to use segments on private logical volumes must explicitly attach them. See public.

protect

A lock protects a data base or data objects, and/or operations on it, if such operations on that object or data base cannot be undertaken unless the process attempting to do so has the lock locked. For instance, the AST lock protects deactivations.

pseudoclock

A counter that is incremented every time an event occurs. Via appropriate protocols, an old value of a pseudoclock may be saved, and compared with a new value, an equal comparison implying that no occurrences of the event have happened.

PTW

For page table word. A processor hardware control word, an element of a page table, that specifies either a main memory frame address or that the processor should take a page fault when attempting to use this PTW.

PTW-level abs-seg

An abs-seg implemented via a page table; the SDW for this segment number describes that page table, and the PTW contents and PVT index in the ASTE are varied to describe the extent of disk or bulk store being addressed.

public

Of a logical volume. One on which access to segments is restricted solely by the ACLs on the segments. See private.

pure
or purify

A pure page is one that has a good (i.e., identical) copy on secondary storage or the paging device. To purify a page is to write it out so that this is so.

PV hold table
or PVT hold table

A table of half-locks, protecting nonunitary VTOC operations against the physical volume demounter. Also used to schedule salvages if crawlouts occurred with a volume "held" (half-locked).

PVT

See physical volume table.

PVTE

For physical volume table entry. See physical volume table.

quota

An administrative limit on disk record consumption. The quota of a directory is the maximum number of nonzero or in-main-memory pages allowed to be created for segments charged to the quota account of that directory.

quota account
A data structure associated with a directory (in the VTOCE and/or ASTE of that directory) that allows segments inferior (not necessarily immediately inferior) to that directory to have their record consumption charged against a single pool. See quota.

quota cell
The information in an ASTE or VTOCE for a directory with a quota account that describes the quota limits and records currently charged against that quota account.

read-write sequence (RWS)
A sequence of operations by means of which the copy of a page on the paging device is written back to a record of disk. This is only performed for pages for which the paging device copy is different than the disk copy. An RWS consists of allocation of a main memory frame, reading in the page from bulk store, and writing it to disk, freeing the frame when done.

record
1. (Disk record) A 1024-word, contiguous extent of disk that can hold a copy of a page.

2. (PD record) A 1024-word, contiguous extent of paging device that can hold a copy of a page.

repatriation
The act of locating the segment to which pages "trapped" on an unflushed paging device belong, and writing these pages back to the appropriate disk record. Performed by the post-crash PD flush.

residue
The data left over in a record of disk or bulk store or a frame of main memory after a given page no longer resides there. It is impossible to read residues.

resource control package (RCP)
Multics subsystem (running in ring 1) that controls and mediates access to peripheral devices. RCP also controls the attachment (see attached) of private logical volumes to user processes.

resurrection
The act of converting a nulled address into a live address, which allows it to be reported to a VTOCE file map. See nulled. Resurrection is performed upon the successful completion of a write, double write, or RWS.

reused address
A disk address simultaneously in use by two different pages. Such a situation is theoretically impossible. See unprotected address.

reverse connection failure
A situation that exists when a VTOCE describes a segment, but no branch in any directory in the storage system describes that VTOCE (or therefore that segment). Such a segment is said to be an orphan. See connection failure.

reverse-deciduous
A segment in the storage system hierarchy that is placed into the hardcore address space of some or all processes via semipermanent activation. Examples are the PDS of any process except the initializer and >online_salvager_output. See deciduous.

RLV
See root logical volume.

root logical volume (RLV)
The logical volume, which contains the RPV, on which all directories exist. It is the only logical volume necessary for system operation.

root physical volume (RPV)
The disk, residing on the drive pointed to by the root config card, on which the root directory (>) and the hardcore partition (in which the supervisor resides) exist. It is a member of the root logical volume.

RPV
See root physical volume.

RPV-only directory
A directory, whose sons logical volume is the RLV, whose inferior segments and directories may only be placed on the RPV. The root is such a directory, as is >lv.

RPVS
BOS keyword for root physical volume salvage. A volume salvage of the root physical volume (RPV), that is performed by the system during initialization when necessary or requested. Most cases of RPVS also involve some automatic directory salvaging. See short RPVS.

run
Said of disks, the bulk store, or their DIMs. To call the appropriate hardware interface modules for a device, and see if operations have completed, invoking the interrupt side of page control when this has happened. One can "run" a device in a loop until an arbitrary number of operations or an arbitrary operation has completed. Simulates an interrupt, in effect.

RWS
See read-write sequence.

RWS abort
The action taken by the page fault handler when a page fault is taken on a page from which an RWS is in progress. When the RWS is posted as complete, the page fault will be resolved by the interrupt side of page control.

scrap
Of paging device records on an unflushed paging device. To cause the system to ignore the contents of such records, forgetting the fact that they are unflushed and in need of repatriation.

SDW
For segment descriptor word. A hardware control word, an element of the descriptor segment, that gives the absolute address of an unpaged segment, or the absolute address of the page table of a paged one. The SDW also contains access mode and ring brackets, as well as other information. The SDW can also specify taking a "segment fault."

SDW-level abs-seg
An abs-seg implemented as a variable SDW slot; various SDWs either describing main memory or page tables are inserted in that slot to describe the main memory extent or segment to be addressed.

secondary storage
Permanent storage as opposed to the paging device. Interchangeable, in most contexts, with "disk."

segment fault
An exception condition detected by the processor (the appending unit), when an attempt is made to use an SDW that describes a segment not yet connected to the process in whose descriptor segment the SDW appears. This is indicated by the bit sdw.df being off. A segment fault causes a specific fault vector to be unconditionally executed, ultimately invoking the segment fault handler.

semipermanent activation
Activating a segment in such a way that it will remain active even after the AST is unlocked. This is performed by the program grab_aste, and is done by turning on the bit aste.ehs, the "entry hold switch."

service

An action performed by a subsystem on behalf of some other subsystem, a class of action so provided. The services performed by a subsystem are its reason for existence. See function for comparison.

setfaults

An operation performed by the procedure of the same name, at the time a segment is deactivated or its access attributes are changed. This operation modifies or faults all of the SDWs for a given segment, located via the trailer list. The associative memories of all processors are always cleared as the last step of a setfault.

short RPVS

A root physical volume salvage (see RPVS) performed automatically upon bootload after a successful emergency shutdown. It is called "short" because no directory salvaging of any kind is performed in this case.

simplex wait protocol

The technique used by the page-fault handler and the multiplex wait protocol to await the occurrence of a page control event. This technique involves the assumption that the "occurrence" of the event may not have happened when indicated, and the status of the operation being performed having to be reevaluated.

sons logical volume

Of a directory. The logical volume on which all segments created inferior to this directory will reside. A directory inherits its sons logical volume from its parent unless it is a master directory.

SST

For system segment table. A supervisor segment (sst_seg) that contains almost all page control data bases, all AST entries, and many meters. It is contiguous in main memory (unpaged), as it contains page tables used by the hardware. See core map, PD map and AST.

temp wired

1. Temporarily wired, via calls to the wiring interfaces (pc_wired) in page control.

2. Sometimes used to mean paged and wired, as opposed to unpaged. See perm wired.

trailer

An entry in the system trailer segment (str_seg) attesting to the fact that a process has an SDW for a given active segment. Each active segment possesses a trailer list of such processes. The trailer identifies the process via the AST offset of its descriptor segment's ASTE, and contains the segment number of the segment in that process. See setfaults.

trickle

See AST trickle.

unflushed

Said of the paging device, or an instance thereof. Containing pages from a previous bootload, that are in need of repatriation. See active for comparison. No new pages are migrated to an unflushed paging device, and the system will not come up if it has one.

unique ID (UID)

A 36-bit number assigned to a segment at the time it is created. It is different from any other UID for any other segment in that hierarchy. It is stored in the VTOCE, ASTE, KSTE, and branch for a segment, and must match for all of these objects. UIDs are also stored on the paging device to facilitate repatriation.

unitary
or atomic operation

An operation performed upon a data base or data object by a process, in such a way that no other process attempting to perform or succeeding in performing the same or other operations upon that data base can affect the operation being performed in any way.

unprotected address

A disk address in use by some page of some segment that is not marked as in use in the bit map for that volume. Such a situation is theoretically impossible.

used list

See PD used list, AST used list, and core used list.

"Used lists" in Multics are circular, double-threaded lists of similar objects, containing both free and in-use objects. All of the free objects are maintained at the head of the list.

Used lists generally implement replacement algorithms, with the entries at the head of the list that are not free the most likely candidates for replacement.

vacate

1. Of a physical volume. To drive all of the segments on a physical volume onto some other physical volume in that logical volume. Done by the sweep_pv command.

2. Of a main memory frame or PD record. To evict any page from that frame or record, such that the frame or record becomes free.

volume header

The first eight records of a physical volume, containing the label, the volume map, the VTOC header, and the bad track list.

volume map

A data base in the volume header of a physical volume that describes the paging region of that volume, and includes a bit map telling which records are in use.

VTOC

For volume table of contents. An array of entries (VTOCEs) describing each segment on a physical volume. The VTOC occupies a fixed, contiguous extent at the beginning of a physical volume.

VTOCE

For VTOC entry. A disk-resident data object that describes one segment on the physical volume on which it appears; this description includes record addresses and other attributes.

VTOC header

A data base in the volume header of a physical volume, that tells the extent of the VTOC, and contains the head of the free VTOCE thread.

vtoce-parts

One of the three physical 64-word parts of a VTOCE. The first vtoce-part contains the activation information, the third the permanent information. The file map is in all three.

wire

Of a page or segment. To make that page or segment wired.

wired

1. Of a page. A page that may not be removed from main memory. The bit ptw.wired tells page control not to replace this page.

2. Of a segment. A segment having some or all of its pages wired. See also abs-wired.

withdraw

To withdraw an address from the free pool of records of a given physical volume (bit map) is to request an unused record and mark it as used, obtaining the address of that record. (Also said "address withdrawn against a given bit map.") See bit map and deposit.

writebehind

A feature of the main memory page replacement algorithm whereby the basic path of the algorithm skips (see preseek) writing, and this writing is done later (at the end of page fault processing).

APPENDIX A

CHANGES FOR MR 6.0

This appendix describes storage system implementation details that are markedly different in Multics Software Release 6.0 from descriptions found elsewhere in this manual. Areas affected, and described in this appendix, are:

1. prewithdrawing policy
2. per-process hardcore segment policy
3. volume dumper support
4. page posting queue
5. page control traffic control interface
6. page control consistency strategy
7. page control error strategy
8. large volume map space
9. damaged segments
10. quota validator
11. support of hierarchy salvager
12. limited update backlog
13. partial shutdown

PREWITHDRAWING POLICY

The algorithm given in Section IV under "PDS and KST Management" for prewithdrawing segments is incorrect. Step 4, if it causes a segment move, causes all the prewithdrawn addresses to be deposited during the segment move. Instead of the conjunction of bits aste.dnzp and aste.ehs indicating "don't deposit nulled pages" (5.0 policy), a new bit, aste.ddnp, indicates precisely this. This bit inhibits reporting of nulled addresses for deposition in pc$get_file_map and pc$list_deposited_add. Thus, segments with this bit on in the ASTE do not get nulled addresses reported to segment control, even after truncation. What is more, the segment mover copies this bit into the new ASTE of a segment move before copying the data. Since nulled pages have disk addresses at the time the segment mover attempts to copy the data, it copies, and thus prewithdraws, nulled pages against the new segment, exactly as desired.

The bit aste.ddnp is seen by the AST replacement algorithm as an entry-hold switch; it is metered against steps-ehs. Thus, no segment with this bit on can be deactivated. However, it can be segment moved, and its ddnp-quality preserved. Thus ddnp implies that the segment ought not to be deactivated because of the prewithdrawn quality of its addresses, as opposed to someone sequestering the page table address, which is the normal reason for setting an entry-hold switch.

When ddnp segments are released from the entry-hold state, the ddnp switch is turned off so that addresses of the segments may be reported and deposited after truncation.

Notice that ddnp does not imply dnzp or vice versa; segments with prewithdrawn addresses can benefit perfectly well from not having zero pages written out/read in, but just created in place.

The entries grab_aste$prewithdraw and grab_aste$release_prewithdraw are used to prewithdraw segments (turning on ddnp) and turn off ddnp. In fact, the existence of ddnp simplifies grab_aste radically. This routine now uses this bit (turning it on only to turn it off later, if not the $prewithdraw entry) to force a segment into a sufficiently large ASTE. The algorithm given in Section IV under "Semi-Permanent Activation" is no longer necessary. By turning on aste.ddnp, and touching the needed page, without storing into it, grab_aste can be sure that the segment cannot be deactivated as long as aste.ddnp is on. When a boundsfault occurs, boundfault moves the page's PTW as well as the bit aste.ddnp, causing the page not to go away until the bit is turned off.

PER-PROCESS HARDCORE SEGMENT POLICY

The descriptor segment of a process is now a reverse-deciduous segment. So are all PRDSs, except the bootload-time PRDS, which is deciduous. This has the effect of eliminating parasitic segments (see Section VII, "RPV Parasite Segments") for all cases except scratch segments used by the volume salvager and disk rebuilder. Thus, in release 6.0, descriptor segments appear in the hierarchy, in the process directory. They are prewithdrawn as are PDSs, as described in "PDS and KST Management" in Section IV.

The motivation for doing this was to eliminate the need for the "short RPVS" performed automatically after a successful ESD. The need for the short RPVS (see Section VII) was engendered by addresses withdrawn from a volume map but neither deposited nor reported to a VTOCE (for deletion by normal means) at shutdown time.

Thus, the pages occupied by descriptor segments at the time of a successful emergency shutdown are deposited when the old PDD is deleted by delete_old_pdds. This places descriptor segments on any packs where process directories go, as opposed to constraining them to the RPV. Although this has the effect of diluting the I/O load of the RPV, this subjects segment control (setfaults in particular) to the vagaries of many disks.

In release 6.0, therefore, there is no "short rpvs". The volume salvager and the disk rebuilder use the PV hold table (See Section XIV) to cause full volume salvaging of the RPV if the system should crash while their parasitic temporary segments are in use.

PRDSs for all configured CPUs are created and entry-held (and prewithdrawn) for all configured processors at bootload time, by tc_init.

The program "plm", which was used to create parasitic segments, no longer exists. Descriptor segment initialization logic was moved into act_proc, creator of processes.

VOLUME DUMPER SUPPORT

Unlike hierarchy backup, the new volume backup facility is an integral part of the supervisor. Volume backup accesses segments directly via their VTOCEs, avoiding the overhead of scanning directories to seek out and initiate these segments. There are several important facilities of the volume dumper in segment control, and several important ramifications of its existence.

The volume dumper maintains in the label area of each pack (see Section XIII in the "VTOC HEADER" records, #4 and 5), a bit map of segments that have been modified since dumped. When a volume is accepted, load_vol_map causes (via the program dbm_man) a region to be allocated in the global hardcore segment dbm_seg to contain this map. It is read in, and a pointer to it left in the PVTE for the volume. When the label of the volume is written back, so is the "dumper bit map".

When the volume is in use, all primitives that modify or detect modification to a segment or a VTOCE (notably pc$get_file_map, truncate_vtoce, create_vtoce, and delete_vtoce) call an entry in dbm_man to set the bit corresponding to that VTOC index (the dumper bit map is indexed by VTOC index).

Like record address depositing, setting dumper bit map bits is an operation that must be protected from volume demounting in the window after segment modification has been noted. Thus, the demount protection brackets (described in Section XIV under "Demount Protection") protect dumper bit map setting as well. Since this is now done in create_vtoce, this program now uses the demount protection-bracket mechanism where it did not before, and thus the unitary quality of vtoc_man$alloc_and_put_vtoce is no longer necessary.

The incremental volume dumper scans the dumper bit map to locate segments to be dumped, turning off the bit once they have been dumped. The volume dumper dumps segments and directories in the same way: it dumps binary images. (The volume dumper does, however, lock directories by UID--anonymously, i.e., no pointer--when dumping them, in order to get a consistent copy of the binary object, i.e., no one should be modifying it.)

The volume dumper (incremental, consolidated, or complete) accesses segments via a special entry to "activate", which activates a segment given its PVID and VTOC index, without its branch. This "parentless activation" is performed only for the volume dumper. When the volume dumper wishes to activate a segment for dumping, activate first hashes it into the AST (as for any activation) to see if it is already active, and returns the AST entry pointer if it is active. When activate so does, it turns on the "dumper in use switch" (aste.dius) to prevent any other process from deactivating the segment (get_aste knows to skip such segments). If the segment is not active, activate activates it again setting aste.dius. Any other attempt to activate this segment finds this ASTE, as it is hashed in normally. The bit aste.dius is not turned off until the dumper is finished. When parentless activation is accomplished for the volume dumper, quota checking is suppressed (aste.nqsw turned on) so that page control does not chase up a nonexistent parent pointer.

When the dumper finishes dumping a segment, it must deactivate it if it activated it, and the segment has not yet acquired a parent pointer.

The dumper uses the hardcore segment number of "backup_abs_seg" to construct abs-segs to reference the segments it activates by the above means. It puts itself on the trailer of the segment (see "Trailers and Setfaults" in Section II), such that if the segment is deleted while the dumper is dumping it, the dumper takes a fault, and cleans up. The various entries in trailer management are ccgnizant of the possibility of a trailer with a hardcore segment number for this purpose.

The volume retriever operates by using the standard VTOCE/segment creation primitive (create_vtoce) to create new segments, creating the VTOCE and segment for an extant branch if there is one (forward connection failure). If a VTOCE (and thus segment) already exists for a segment being retrieved, it simply copies the new data into it. Directory contents are merged (see the program retv_copy) in ring zero. When the volume retriever is called upon to retrieve a segment whose branch does not exist, a special entry in directory control's "append" primitive is called upon to create a branch from saved binary information, as opposed to user-supplied symbolic data, without regard to access checks. In this case, the VTOCE/segment being retrieved is connected to this branch.

The volume reloader is not part of the supervisor; it constructs complete physical volumes from volume backup tapes, placing VTOCEs and segment contents on it as appropriate. It uses user-ring disk I/O, and works on volumes not now in use by the storage system.

The segment adopter, and the -adopt option to sweep_pv construct a directory branch for a segment whose VTOCE is extant, but has no branch. As such, it is not part of segment control. The primitive used by both is the same entry to directory control's append primitive used by the volume reloader to construct a branch for an item whose VTOCE (and actual data) it is retrieving.

PAGE POSTING QUEUE

Page control posting strategy (see "I/O Posting" in Section VIII) has been modified to make it no longer necessary for the disk DIM interrupt side to loop-lock the global page table lock. On multi-cpu systems, where the disk DIM interrupt side (on a real interrupt, as opposed to a run) can find the page table lock locked, this loop locking consumed a substantial share of system resources prior to release 6.0.

The solution to this problem was to construct a queue of coreadd/errcode (the parameters to page control supplied by the disk DIM in a posting call) pairs that the disk DIM wanted to report to page control, but rather would not, because the page table lock was locked at the time. Any program at all that unlocks the page table lock is responsible for checking out this queue, and calling page$done_ (see Section VIII) with each coreadd/errcode pair in it.

This queue is called the "disk posting delay queue", or the "coreadd queue", due to its content, and resides in the segment "disk_post_queue_seg". The locking policies involved make sure that everything that is put in the delay queue is processed as soon as possible, and that no requests are lost are quite involved, and are further described below. The maintenance of the delay queue is all performed in the program core_queue_man, in bound_page_control.

The locking policy of the coreadd queue may be expressed as follows: The coreadd queue has a lock on it (in disk_post_queue_seg), which must be loop-locked. This lock is higher than the page table lock (see "Locking Hierarchy" in the glossary). Absolutely no lock-looping is performed with the coreadd queue locked (i.e., no lock is higher than the coreadd queue lock). The only code in the entire system (during normal operation, i.e., not ESD) which unlocks the page table lock is in core_queue_man, and does so only while the coreadd queue is locked. Thus, before actually unlocking the page table lock, a potential unlocker of the page table lock is in a position to inspect the coreadd queue. If the queue is empty, the page table lock can be unlocked and then the coreadd queue lock can be unlocked. If there are posting requests in the coreadd queue, the first one must be taken out, the coreadd queue unlocked (the page table lock is still locked), and call page$done_ while the page table lock is still locked, to perform the posting. It is then necessary to try again to unlock the page table lock, starting by locking the coreadd queue lock. Among the unlockers of the page table lock are page$done, called by the disk DIM interrupt side, in the case where it did not initially find the page table lock locked (i.e., could lock it), and has called page$done_. An attempt must be made to lock the page table lock before the coreadd queue is unlocked when a process has locked it to queue a request for posting there; otherwise, the request might stay forever in the coreadd queue if the page table lock indeed became unlocked between the time the interrupt side found it locked and the time the interrupt side locked the coreadd queue lock (once the coreadd queue lock is locked, the page table lock cannot be unlocked except by the process which has it locked). If the attempt to lock the page table lock is successful, all postings must be done by this process. If not, the process can rest assured that whoever has it locked is going to have to unlock it, and can't possibly unlock it until the current process unlocks the coreadd queue lock, which it has locked, and thus, that other process will find the request just queued as soon as it unlocks the page table lock.

The coreadd queue must be processed at "run" time, and flushed at ESD time as well.

See core_queue_man.alm for more information.

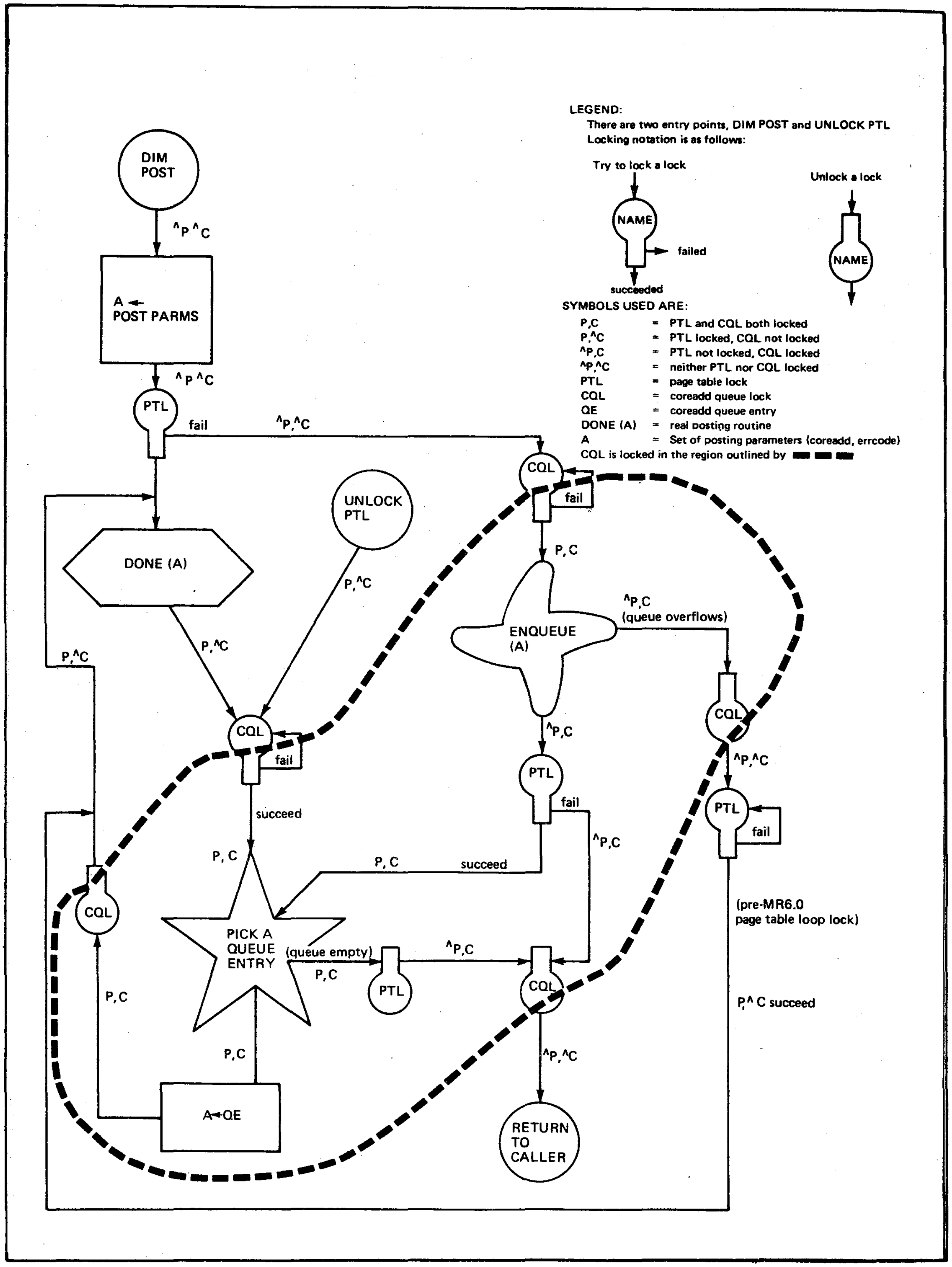

Figure A-1. Coreadd Queue Locking

PAGE CONTROL TRAFFIC CONTROL INTERFACE

The implementation of the disk posting queue involved cleanup in page table locking and unlocking. The unlocking of the page table lock under protection of the traffic controller lock (Section VIII under "Stack Management and Interface with the Traffic Controller") is no longer done. In release 6.0, page control unlocks the page table lock before the traffic controller lock is locked, when going to wait. Taking advantage of some features of the new lockless scheduler, page control does a standard "addevent" when it is going to branch to the traffic controller, storing a wait event (which it knows has not yet been notified, this decision made under the protection of the page table lock, under which all page control notifies are done) in the APT entry of the waiting process. If the traffic controller finds, under the traffic control lock, that this event has been notified (become zero), the traffic controller returns to page control to restart the fault or call side operation.

These changes allowed a new mechanism for waiting for the page table lock from the call side to be implemented. When the call side of page control attempts to lock the page table lock (in device_control.alm), a branch is taken to the traffic controller for page-table lock waiting if it cannot be locked. By the identity of the entry point called, as encoded in the value of pds$pc_call (as for waiting for paging events), the traffic controller returns to device_control$dvctl_retry_ptlwait to reattempt to lock the page table lock when it has become unlocked.

Thus, the only times that the page table lock is looped on are at process-loading time, and if the coreadd queue is full.

Page control no longer uses regular traffic-control waiting for the page table lock; a special traffic controller state is used exclusively for this type of waiting. Also, traffic control returns to a point in the page fault handler instead of restarting a page fault when ptlocking-waiting is complete, in order to avoid the fault overhead. This relies on the fact that the page fault data stored in the PDS cannot have possibly changed since the page fault was taken.

Traffic control no longer needs to validate page control events under the traffic control lock (described in Section VIII under "Global Page Lock"). The above interface wherein page control stores wait events directly in the APT entry of a waiting- process (even during the process loading function) obviates the need for this validation. If an event becomes invalid, a notify clears it out of all APT entries.

PAGE CONTROL CONSISTENCY

Until release 6.0, emergency shutdown (ESD) has been a fairly risky proposition, because of the unknown state of page control data bases at the time the system crashed. Whether or not the system crashed in page control, or because of some problem detected in page control, there was not (and is not now) anything to prevent the system from crashing when some CPU was in page control or the disk or bulk store DIM. This is critical because ESD relies on the correct functioning of page control, not only to write out pages of segments, but to support the virtual memory in which much of ESD runs. The assumption that page control could be used reliably after a crash was therefore not always valid: inconsistently threaded data objects could often cause faults to be taken, and I/O requests in the process of being queued or posted often get lost, causing the system to hang indefinitely awaiting their completion. At worst, these inconsistencies led to misrouting of data, and most often to failure of emergency shutdown in one way or another, with all concomitant grief.

Thus, all data and state manipulation in page control was redesigned and reimplemented to make the following statements true at every point (at all times):

1. If page control is interrupted at this point, a procedure running at ESD time can compute distinctly, fully deterministically, a valid state of the entire data base of page control, reflecting its state either before or after a database change that was interrupted completed or would have completed.

2. If page control (or the disk or bulk store DIM) is interrupted at this point by a system crash, a procedure running at ESD time can regenerate any I/O that was queued, in progress, or in the process of being queued, posted, or performed, without fear of the original I/O ever being posted.

The "procedure running at ESD time" is pc_recover_sst in bound_page_control, also well worth time reading. This procedure places the entire page control data base (the SST) in a consistent state before any paging or page control operations are attempted by ESD.

The fundamental truth that allows this technique to operate is that very little of page control is actually changing the data base, or therefore, the state of page control. Most of page control is making decisions, and calling subroutines. It is only at the very lowest level, almost entirely in ALM page control that the data base is changed. Most of PL/I page control is simply making decisions and mapping the actions of ALM page control over segments. Thus, in order to recompute the consistent state of interrupted page control, we need not know what decisions were being made, or what segment-wide operations were being performed. All low-level page control operations involve only one page of one segment; when one page replaces another in memory, this is really two operations: an eviction and a paging-in. Between the two operations, the main memory frame is distinctly free. During the eviction, or during the paging in, the page under consideration is either in main memory or not: there is no inconsistency involving two pages. Other page control operations are comparably defined.

Typical of the operations under consideration that may be interrupted and must have their state recomputed are:

1. Binding a page of a segment to main memory (paging-in),
2. Unbinding them (eviction),
3. Binding a frame of PD to a page of a segment (PD Migration),
4. RWS initiation,
5. RWS completion, and
6. Unbinding a page from a PD record, either at RWS completion time or during PD Housekeeping.

Each of these operations involves the establishment or revocation of bindings between at most one page of one segment, one main memory frame, and one PD record. As a matter of fact, each such operation consists of the establishment or revocation of at most one (usually bilateral) binding. Each such bilateral binding is usually two values that designate each other. For instance, the binding between a page of a segment and a page of main memory is expressed by the fact that a PTW has a main-memory type address in it, designating a CME that has the address of the PTW in it. The binding between a page of a segment and a record of paging device (PD) is expressed by the page of the segment (PTW pointer) being in the PD map entry, and the PD address being in either the PTW or CME, depending on whether or not the page is in main memory. During a Read-Write Sequence (RWS), a similar bilateral binding between a PD record and a main memory frame exists in crossing pointers in the CME and PDME involved. Therefore, the establishment or revocation of any binding involves, in essence, the setup of two (perhaps conceptual) pointers. Bindings of objects are never changed (except in one case in evict_page, which is quite special) from "bound to this" to "bound to that", but only from "free" to "bound to this" or vice versa. Thus, every page control object can be viewed as "bound to something" or free at any instant, by looking at some critical pointer or field in it. For instance, if a CME has a nonzero cme.ptwp (or mcme.pdmep), it may be considered to be bound to that page of a segment (or PDMAP entry during an RWS), or else none. If a PTW has a main-memory type devadd in it, then that page is bound to that frame of main memory, or else none. The presence of a PD-type devadd in a PTW or CME (which itself is bound to some page (PTW)) says that that page is bound to that PD record, or else none. The presence of pdme.used in a PDMAP entry says that that PD record is bound to the page whose PTW is designated by pdme.ptwp, or else none.

Thus, certain critical fields determine distinctly, at any real time instant, whether or not a given object is bound to some other kind of object (and if so, which one). Before an object is marked a bound to some other object, all other fields except the critical field are filled in to their final values. If page control is interrupted before the critical field is filled in, pc_recover_sst finds the critical field not filled in (usually zero, see last paragraph), and the noncritical fields are essentially garbage; the binding is considered not to have started at all. If the critical field is found filled in, all other fields must be valid, and the binding was entirely complete.

The problem is therefore reduced to consistency between halves of a bilateral binding. This is accomplished by simply stating an order in which halves of bilateral binding are accomplished, the unbinding being accomplished in the opposite order. Thus, if pc_recover_sst finds two valid bindings, which are halves of a bilateral binding, the entire bilateral binding must be complete. If it finds one half of such a binding complete (after determining completeness by the rules of the last paragraph), it can either complete the binding or complete the unbinding, without regard to whether a binding or unbinding was in progress at the time page control was interrupted.

The following rules govern the establishment of bilateral bindings:

Pages to main memory frames, and vice versa:
when binding (reading-in), first bind the CME to the PTW, then change the PTW to designate main memory. When evicting, do the opposite.

Pages to PD records, and vice versa:
When binding (PD migrating--always happens when page in main memory), first bind the PDME to the PTW, and then change the CME to the PDME. When performing PD eviction, either at the completion of an RWS or a pure eviction during PD housekeep, do the opposite (i.e., first change the PTW or CME, then free the PDME). At all of these times (migration, RWS complete, pure eviction, and in-core PD eviction in pc.pl1) the copy of the page on disk or in main memory is, or is the same as, the most recent.

PD records to main memory frames, and vice versa (during RWS):
First bind the PDME to the CME, and then the CME to the PDME. At RWS complete time, do the opposite.

The handling of I/O in progress at the time of the crash is made trivial by the action of page$esd_reset, which calls entries in the disk DIM and the Bulk Store DIM to throw away the entire contents of their queues, and reinitialize their data bases. Thus, any page that is seen as out-of-service by pc_recover_sst may be simply evicted if it was a read in progress, knowing that the read is not actually in progress (the system crashed), and is not posted (the queues are flushed). If a write was going on, the modified bit is turned back on when this is done, because the action of initiating the write caused the modified bit to be turned off by write_page (the latter knowing that the page would be written). The modified bit is not turned back on, however, a page that is being updated as pure ("nypd write") to the paging device. The bit cme.pd_upflag, reclaimed for this purpose, indicates during a write that this is the case.

The routine pc_recover_sst can tell if the above rules have been violated, due either to bug, processor or memory malfunction, or damage to the page control data base by other parts of the operating system. Even in this case, it attempts to make the page control data bases consistent so that ESD can succeed. When such unexplained damage (i.e., inconsistency that cannot happen by virtue of the above rules) is detected, segments are marked as damaged and involved pages zeroed where appropriate.

The flushing of DIM queues at ESD time substantially simplifies the ESD strategy of the VTOC manager (see Section III, "ESD Strategy"). The VTOC manager can now decide distinctly that no I/Os queued before the crash are ever going to be posted. The bit b.ioq is now superfluous.

PAGE CONTROL ERROR POLICY

Release 6.0 makes radical changes to the handling of disk errors as detected by page control. First of all, errors are not reported to the operator console or the syserr log unless a page is actually damaged. The disk DIM has already reported all device error information for any I/O operation involved. A differentiation is made between device errors that affect a particular record gone bad, and those that are an indication of a device problem. In the latter case, it is almost always true that the operator can re-ready the device, or it will re-ready itself, or some nonautomatic remedial action can be taken. Thus, in any of these cases, it is unwise to perform irreversible action such as damaging a segment, or even wasting syserr log space with messages. The disk DIM differentiates between the device error case and other cases in the value of the error code at posting time. Errors reading either therefore replace the disk address in the PTW with a null address or not (as the disk error was a per-record error or a device error) before setting ptw.er. When such a page fault is restarted, a successful page fault either pages in zeros or the correct page, respectively.

Write errors determined to be due to an inoperative device cause the posting to cause the modified bit of the page to be turned back on (disk writes only--bulk store cannot be inoperative by this standard), and the core frame to be threaded back in as MRU. This means that the replacement algorithm will reissue the write again when it comes around. If the call side started the write, it calls in again to write it again, as it comes back to see that the page is still modified (or yet modified) when it is notified. Similarly, device-inoperative errors on the write cycle of an RWS cause the PD record not to be freed, but placed back in the PD used list (its modified bit was never turned off), and the free-or-being-freed count (sst.pd_free) decremented. The PD replacement algorithm retries that record at a later time.

The system no longer signals page_fault_error on a read if the cause of the read error is an inoperative device (as opposed to a bad page). This is to avoid signalling errors that might well terminate an absentee process or the initializer in cases where the operator's readying of a disk could allow all software to proceed without error if the supervisor cooperated. Other problematic cases of signalling page_fault_error, such as on a descriptor segment which goes offline during a setfaults operation, are avoided in this way as well.

Instead of signalling page_fault_error, processes that seek to read pages on inoperative devices are made to wait upon a global event, in ring zero, "144163153176"b3, being "dskw" in ASCII, until any disk coming back online notifies this event. The disk DIM performs this notification, and now maintains the bit pvte.device_inoperative, previously used only for drive-test operations, as a copy of its "broken" bit for a given device, notifying this event whenever such a bit is turned off. Any time such a bit is turned on the disk DIM has beeped a "Device requires attention" message to the operator.

The maintenance of pvte.device_inoperative has several implications: when a disk goes off line, the VTOC manager can see that at once, and reject a requested write forthwith, without wasting hot VTOC buffers where not necessary. The create_vtoce primitive can avoid creating segments on inoperative devices. More critically, update_vtoce must be prepared to handle error codes from vtoc_man for inoperative disks, realizing that the vtoce-parts requested were not even put in hot buffers. For this reason, update_vtoce$deact now has an error-code argument.

The implementation of this "disk-offline waiting" feature is facilitated by the fact that all callers of page-reading primitives must obtain the event to be waited on from the primitive in question, because volume-map paging issues preclude any other routine from deducing the wait event. Thus, page reading primitives can now return this global disk offline event, and cause any number of mechanisms to wait and retry on this event. There are exactly three interfaces that call read_page: the page_fault handler, the PL/I-side interface page$pread, and evict_page$abs_wire. These primitives all now check for the presence of ptw.er from a previous read before calling read_page. Thus, if a page read error is posted by the "done" side, an immediate notify causes one of those three interfaces to be reinvoked (via repeated page fault or call-side retry protocol), notice ptw.er, and take special action.

This special action consists of calling page_fault$disk_offlinep to determine if the reason for this error is the disk being offline or some other reason. This is determined by inspecting the PVTE bit set by the disk DIM (there is a window here--it might have been inoperative at one time, but operative now--this is acceptable). If the answer is that the disk is offline, the process page-faulting, call-side (or process-loading-side) reading, or abs-wiring is made to wait on the global disk-offline event. The bit ptw.er is turned OFF at this time, before the process is set waiting, so that when the disk comes back online, a retry of the page fault/reading is made as though no error happened, instead of the detection of the previously set error bit (which this time would be guaranteed to find the disk not inoperative, and thus signal, which is precisely what we are trying to avoid).

If page_fault$disk_offlinep determines that the disk is not offline, an alternate return is made. The page fault handler signals in the way it always used to in this case, and the other two entries just retry desperately and hopelessly as they used to do. (This is the case of a descriptor segment page going bad or similar--an unsolved problem as of this time.).

The call-side wait coordinator, and the notify-requested bit setter in wired_plm (process loading) have been made cognizant about global paging events.

LARGE VOLUME MAP SPACE

In releases 4.0 and 5.0, the single paged unwired segment fsdct held all volume maps. This was an unreasonable space limitation. Volume maps are now in segments fsmap_seg0 to fsmap_seg15, created dynamically by init_pvt at bootload time, as many as are necessary to contain the volume maps for all configured drives. The segment fsdct now contains only what used to be the fsdct header; it is small, unpaged, and wired now.

The code in free_store that returns a PTW pointer and an ASTE pointer to read_page now deduces these quantities from the SDWs of the fsmap_seg, rather than from a fixed pointer in the SST.

Therefore, all references to "FSDCT Paging" in this document should now be read as "Volume map paging".

DAMAGED SEGMENTS

A new VTOC attribute (see Section II, "VTOC Attributes"), thus an ASTE and VTOCE bit, called the "damaged switch", has been introduced (aste.damaged and vtoce.damaged). Although settable and resettable by user-invoked file system calls, the intended function of this bit is to inform the user that page control or the physical volume salvager has either perpetrated or detected damage to this segment. The segment fault handler observes this bit when connecting a process to an ASTE (i.e., constructing an SDW for a segment in a process), and causes "seg_fault_error" with the error code of error_table_$seg_busted to be signalled if it is on. As with other VTOC attributes, the bit is activated and deactivated with the segment., The segment fault handler does not make this check for directories, or in the initializer process (so that the system might always be bootable).

The physical volume salvager and page control construct a standard format binary syserr message (see segdamage_msg.incl.pl1) whenever damage to a segment is created, and log a message with it. This message identifies the segment involved via physical volume ID, LVID, UID, and UID pathname, with other information (e.g., page number) when appropriate.

The physical volume salvager constructs this information from a VTOCE being processed, the UID pathname being copied from the third vtoce-part. Page control deduces it from AST entries, chasing up the AST parent pointers to develop the UID path (this logic is in the module page_error). The physical volume salvager "damages" segments whenever any VTOCE inconsistency is discovered: the case where segment control deliberately introduces an inconsistency during VTOCE update before a fatal crash is particularly important here. Page control damages a segment when a disk error on reading or writing occurs that is due to a bad record as opposed to a bad device.

The counter sst$damaged_ct is incremented whenever such a binary message is logged. The answering service's accounting-update metering program (as_meter_) inspects this variable at each accounting update. If it has increased (since the last update, or bootload time, initially), the syserr log is scanned for such messages. They are read out, the UID pathnames in them converted to ASCII pathnames, and the interpreted messages logged in the answering service log.

QUOTA VALIDATOR

Reimplementation of what had been the salvager in release 5.0 and earlier, for this release removed the function of computing quota-used from it. Quota-used computation was the only part of the salvager that could not be done by a top-down hierarchy scan; one cannot compute correct quota-used for a directory until correct quota-used totals have been computed for inferior directories; this severely limits the implementation flexibility of salvaging functions. What is more, the algorithms up to now for correcting quota-used required the entire hierarchy to be quiescent: thus crashes for which ESD has failed (almost guaranteed to create quota-used inconsistencies see below), required a "long salvage" while no one was logged in (the only way the salvager could be run).

The discovery of an algorithm to compute correct quota-used totals in a nonquiescent hierarchy has obsoleted all of this, and is now the only way that quota-used is corrected. The hierarchy salvager is now nothing more than a program that reformats a single directory, optionally cross-checking VTOCEs. Conventional ring-4 programs are used to map the salvager over subhierarchies. Quota and quota-used are now out of its domain.

In order to understand the on-line correction algorithm, it is necessary to understand how quota-used inconsistency arises. A subhierarchy is said to have inconsistent quota-used if any directory in it has inconsistent quota-used. A directory is said to have inconsistent quota-used if its quota-used figures (for segments or directories) are anything but what they should be. The (directory or segment) quota-used figure of a directory should be the sum of the (directory or segment) quota-used figures of all immediately inferior directories that do not have terminal (directory or segment) quota accounts, plus the sums of the records-used of all immediately inferior directories or segments. This is dependent upon all subhierarchies being quota-used consistent.

A directory becomes quota-used inconsistent in the following way: a segment is deleted or some pages are created. Several directories have their quota-used figures adjusted by page control (in the ASTE) at the time this happens. At some later time, the VTOCE for one of the directories is updated; perhaps the lower one is deactivated, or the AST trickle updates one of them. The VTOCEs now reflect an inconsistent quota-used situation, for the VTOCE of one directory claims records charged to it, but the other does not. If the system shuts down successfully there is not problem, as all VTOCEs are updated. Before the system shuts down, anyone who wants to know the quota-used figures goes to VTOCE or ASTE as appropriate, and the inconsistency of the VTOCEs is not a problem. However should the system crash and not shut down, the next bootload relies solely on VTOCE information, and a quota-used inconsistency results.

It may be seen that quota-used inconsistency is not the result of a supervisor malfunction, but rather a misfeature of fatal (no ESD) crashes. They are a consequence of not stopping the entire system to update disk-resident data every time a page is created or destroyed. Quota-used inconsistencies do not develop while the system is running.

The online correction algorithm is based upon the fact that quota used for a given directory is either right or wrong at any time. If it is right to start with, it cannot go wrong while the system runs. If it was wrong to start with, the amount by which it is wrong is a constant from the time the system was bootloaded to the time it is fixed. It cannot get more or less wrong by its own volition.

The task of the quota validator is thus to determine exactly how much (if at all) a given quota-used figure is wrong and fix it. It can fix it at any time after it determines by how much it is wrong--a certain number is to be added or subtracted. The quota-used figure is not just replaced.

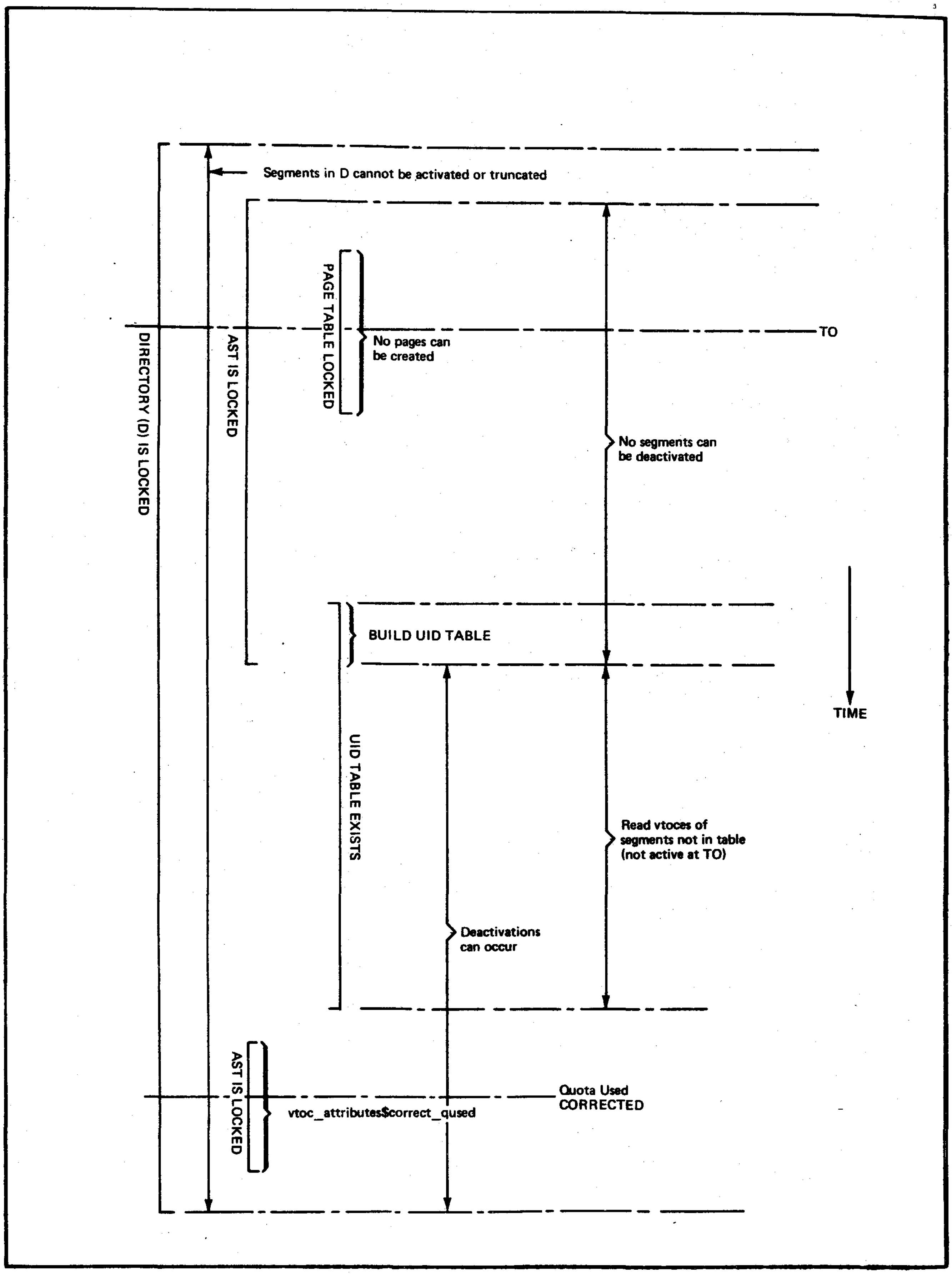

Figure A-2. Quota Validator

To understand this more fully, hypothesize that there were a tool available that corrected quota used, say set_quota_used <dirname>. A system administrator might want to figure out the correct number, and set it. However, this would be inordinately difficult to use, because even the wrong number is constantly changing. Thus, the only kind of tool that would be of value is one that added or subtracted its argument from the quota-used figure, regardless of what it was--a tool that added or deleted phantom segments.

The quota validator operates precisely in this way. The entry vtoc_attributes$correct_qused performs precisely the function of adding a signed difference to a quota-used total for a directory, either in a VTOCE or in an ASTE, once the correct difference has been determined. The determination of the value of this difference is a very intricate operation, involving several locking games. We can approach this algorithm by successive refinement.

Given our choice, we would quiesce the entire subhierarchy of the directory (which we will call D) whose quota-used is being computed. We would lock the page-table lock and the AST lock, read all the VTOCEs and AST entries for immediately inferior segments and directories, adding their page totals and quota-used figures (for directories), from the AST for active segments and from the VTOCEs for nonactive segments. Comparing that total to the current quota-used gives us the difference we seek. However, we cannot randomly go locking locks like that, or quiesce the subhierarchy in this way. We therefore choose one moment in time for which we will strive to compute D's correct quota-used total. For any given instant, we can quiesce all of page control activity (creating and deleting pages of active segments in particular) by locking the page table lock. Call that instant T0. We choose such an instant, and lock the page table lock before it. At that instant, with the page table lock locked, we compute the sum of the records-used totals of segments immediately inferior to D, that subset of them that is active at T0, plus the sum of the quota-used figures of immediately inferior (nonterminal) directories, that subset of them that was active at T0. This figure is an approximation to the correct sum of records-used plus inferior quota-used for this directory at T0. It is inaccurate by precisely the sum of the records-used plus nonterminal quota used of exactly that set of immediately inferior segments and directories that were not active at T0. Thus, once the page table lock is unlocked, we need only add up the figures for these segments. However, we do not wish to read all the VTOCEs, or scan D with the page table locked. If we unlock the page table lock, other segments may be activated or deactivated, and we would have no way or determining which segments were active at T0 and which were not.

Pages are created only by touching them, and since only pages of active segments can be touched, no pages can be created for inactive segments if we prevent them from being activated. Similarly, pages can be destroyed by two means: manipulations on active segments (truncations, page zeroings), and file-system calls (truncate, delete) on inactive segments. Thus, if we prevent new segments in D from being activated between T0 and the time quota-used of D is corrected, and prevent file system operations on segments in D in this interval at well, we can be sure that the quota-used subtotal for segments inactive at T0 will not change between T0 and the time quota-used of D is corrected. It turns out that locking D prior to the start of this whole operation accomplishes precisely this.

With this in mind, we know that no segments that were not active at T0 can be activated after the page table lock is unlocked. What is more, they cannot be otherwise affected (e.g., truncated). So at this stage of development, our algorithm is to unlock the page table, scan D, and check each segment in it for activity at time T0 (it couldn't be active now if it wasn't active then) and add its quota-used or records-used to the total from time T0. This does not work because segments can get deactivated between the time the page table lock was unlocked and the time we check the AST to see if it was counted in the total at time T0. Segments can be prevented from being deactivated by having locked the AST after first locking D, but before locking the page table lock. Thus, when we scan D, the AST will still be locked, and the set of active inferiors of this directory will not have changed since time T0.

The deficiency here is that one may not touch a directory with the AST locked (see the general considerations of the locking algorithm in "Locking Conventions", Section II). To determine which segments were active at time T0 we lock the AST lock before locking the page table lock, and unlock the AST lock after unlocking the page table lock. But before unlocking the AST lock (at a time when the set of active segments cannot have changed since T0), we build a little table of the UIDs of all active inferiors of this directory in automatic storage. It is with this table that we check while scanning the directory adding up quota and records figures from VTOCEs.

Having added the active and inactive figures, they are compared with the value of the quota-used figure of this directory read at time T0 to determine the finite and invariant error. It is this error that is deducted from the quota-useed figure of D.

This algorithm is implemented in the program correct_qused. The entry quotaw$rvq performs the manipulations and quota cell readings under the page table lock.

The bottom-up walking features of do_subtree (or walk_subtree) are used to drive the tool fix_quota_used (the ring 4 interface to the quota validator) bottom-up.

SUPPORT OF HIERARCHY SALVAGER

The mechanism used by the hierarchy salvager to activate, deactivate, and access segments, dating from the time that the salvager had its own tape, is entirely gone in release 6.0. The entire activation/file map mechanism described in "Services on Behalf on the Hierarchy Salvager" in Section IV has been removed. The hierarchy salvager is now a directory-control program that operates on one directory at a time, given its pathname. It initiates directories and takes segment faults upon them, as any other directory control program in Multics. It has no more involvement with segment control. The removed interaction with segment control had been a major source of bugs.

The central control program of the hierarchy salvager, salv_directory, is usually driven by ring-4 subtree walk. It does not recurse.

The hierarchy salvager no longer uses abs-segs or any abs-seg mechanism; it no longer checks, validates, or corrects quota or quota-used.

The hierarchy salvager retains a "VTOCE-checking" feature, used to check for (forward) connection failure, optionally delete branches suffering this, and correct part III (permanent attributes) information. These functions are provided in the program salv_check_vtoce, which is not even called if VTOCE checking was not specified to the hierarchy salvager. The program salv_check_vtoce calls vtoc_man$get_vtoce to obtain a VTOCE image, to check UID match and part III information. If information need be corrected in the VTOCE, the entry "salv_update" in vtoc_attributes is called to correct and write back information to be updated. As usual, vtoc_attributes is cognizant of all rules regarding directory and AST locks for such operations (see Section II). Thus, the hierarchy salvager no longer directly writes VTOCEs in any case.

To delete branches suffering forward connection failures, salv_check_vtoce calls a special entry in directory control's "delentry" primitive, that which deletes branches.

The hierarchy salvager makes use of the grab_aste/prewithdraw mechanism described above and in Section IV to cause semi-permanent activation of its scratch and directory-copy segments.

LIMITED UPDATE BACKLOG

The 6.0 storage system tries to enforce an upper bound on the time the AST trickle takes to circumnavigate each AST used list (see Section II). By placing an upper bound on this time, file map changes cannot stay in the AST (not be reported to the VTOCE) for longer than this maximum time. This is done solely as a hedge against fatal crashes under light load. In these cases, it has often been reported that a segment modified hours before the crash appears empty (all zeros) at the next bootload. This was because of failure to update its VTOCE within a reasonable period of time. in release 6.0, the initializer calls into get_aste$flush_ast_pool with a pool index every accounting update if it has been determined that fewer steps in that pool than the number of entries in it were taken since the last such update (the accounting update routinely inspects meters in the SST). The entry get_aste$flush_ast_pool circumnavigates the specified AST list one entire time, calling update_vtoce on each ASTE whose file map has changed (aste.fmchanged). This fairly expensive action is invoked if and only if load is so light that there was not a reasonable number of AST steps in the last accounting interval.

A similar attempt is made to set an upper bound on the amount of time a page may stay in main memory and not be written out. This, again a hedge against fatal crashes, is to guard against the phenomenon where a heavily-modified page remains in memory under light load, and does not get written out, and appears zero or nonexistent at the next bootload. A page is written out if load is light, i.e., the circulation speed of sst.usedp is slow, and continual use and modification biases the replacement algorithm against writing this page out.

The new entry pc$flush_core, and the new CME bit cme.phm_hedge implement this facility. The entry pc$flush_core is called five minutes before every accounting update (by the initializer) to call page$pwrite on all pages not written out since the last such call. The five-minute interval is to make sure that the accounting update that follows, calls get_aste$flush_ast_pool, is able to report new page creations to VTOCES, i.e., to ensure that writes started complete successfully before VTOCE updating is attempted (see "Address Management Policy" in Section VII for why the VTOCE can't be updated until successful completion of writes is acknowledged). The entry pc$flush_core scans the core map for all in-core pages that need to be written out, and calls page$pwrite, multiplexing activity in the normal page control manner (see Sections VIII and IX). These pages are identified by the presence of the flag cme.phm_hedge. This bit is turned on by pc$flush_core for every in-core page having ptw.phm on, that it is not calling page$pwrite to write out. Page control (page$pread and the "write" side of the interrupt side, page$done_) turn this bit off any time a page is read into this frame, or a successful write is completed from it. Thus, if pc$flush_core finds (the next time it is called) that this bit is still on, it can deduce that this frame had a modified page in it one accounting interval ago, and has not been evicted or written out since. This is precisely the condition for issuing a write for the page in that frame.

PARTIAL SHUTDOWN

Page and segment control primitives called at shutdown time (Emergency or Regular) have been changed to check the PVTE bit pvte.device_inoperative before attempting to update a VTOCE (including calling pc$cleanup), flush a main memory page or initiate an RWS. All drives are tested at the time shutdown is started (earlier still in ESD), in the procedure disk_emergency (in bound_disk_util_wired). By calling the standard drive-testing primitive (read_disk$test_drive, see "Explicit Disk Reading, Writing, and Testing" in Section XIV) the operative/inoperative status of all drives is determined. What is more, the interrupt sides of page control and of the VTOC manager call an entry in disk_emergency which evaluates whether or not to set pvte.device_inoperative whenever they receive a "device-inoperative"-type error from the disk DIM. The program disk_emergency sets the bit only during shutdown; otherwise, the disk DIM maintains it. At shutdown time, disk_emergency also notifies the Operator about disks which went offline during (or before the start of) shutdown.

Thus, during shutdown, all drives not inoperative are completely shut down. The complete shutdown of the RPV is not indicated unless all other drives were shut down; this is to force a hardcore directory salvage and paging device flush on the next bootload. All packs not shut down will be salvaged the next time they are accepted, as is usual.

The code and variables of Emergency Shutdown have been so reorganized that ESD may be attempted any number of times after a partial shutdown, if drives can be brought back up. If the drives have indeed become operative (all drives are tested afresh each time), a completely successful ESD will be attained. Unflushable contents of the paging device and main memory will be kept around until this is the case.

The avoidance of complete shutdown of the RPV causes the next bootload to take cognizance of the unflushed paging device, which is necessary.

OTHER CONSIDERATIONS

In Section VI, cme.devadd now points to the PDME during the entire RWS.

The variable "did" in pxss_page_stack (the ALM page control environment stack frame) has justly and finally been renamed "pvtx", which is what it had meant since release 4.0.

A fairly baroque error-message generating facility has been built into page_error.alm, taking advantage of the new macro processor in the ALM assembler. Incorporated in this facility is the logging of binary syserr messages indicating segment damage. The page_error program includes a system of macros for declaring variables and generating PL/I-like calls automatically, and is worth investigation by those interested in ALM or assembler technology.

In Section VIII, the "second trace facility", or "disk_meters" has been totally removed.

The subroutine cleanup_page is now the only agency in the system (outside of pc_recover_sst, that is a highly special case) that evicts pages. The routines in pc.pl1 have been changed to call it, as page$pcleanup. Consistency required by pc_recover_sst motivated this change.

In Section X, some reorganization of utility subroutines, particularly in pd_util, was performed.

A (privileged) user-callable facility to entry-hold a segment and wire its pages via calls to pc_wired has been provided.

The updating of time-page product to a directory's parent at the time of its deletion was found to be lacking in Releases 4.0 and 5.0. This function was added in delete_vtoce, which, in the case of a directory with terminal quota being deleted, performs several VTOCE manipulations under the AST lock to update this VTOCE-resident quantity from the directory being deleted up.

Reused and unprotected disk addresses, as well as bad VTOC threads, no longer cause the system to crash. Volumes suffering these symptoms are placed in a state (pvte.nleft = 0) where no new allocations can take place on these volumes, and scheduled for salvage (pvte.vol_trouble = "1"b). These new policies are due to a belief in the current stability of the supervisor: that such symptoms can not occur as a result of a software malfunction in the current bootload, but are more likely symptoms of disk malfunction or bad data from a previous bootload.

The "PD Writeahead" experimental feature has been removed.

The disk record allocator has been recoded to be more straightforward: the remnants of older schemes have been replaced by code which has the same effect, but by explicit design.

The disk-reading primitive (read_disk, Section XIV) is now used by volume backup in many processes, and thus can no longer use the unshareable supervisor ASTE (PTW-level abs-seg) read_disk_seg in all processes. It continues to use this ASTE if running in the initializer process, initialization, or shutdown. in any other process, an ASTE is gotten via normal means (get_aste) to use a an abs_seg.

The VTOC attribute array for record quota (aste.quota, vtoce.quota) is redefined as seg_vtoce.usage_count and seg_aste.usage_count, a count of page faults on a segment maintained by page control, for nondirectory segments. A file system call through mhcs_ is available to obtain this VTOC attribute. It is not in hcs_ because the observing of this datum constitutes an AIM write-down path, and discretionary access control to this meter may be desired at some AIM sites.