Chapter 6

PAGE FAULT HANDLING

1. INTRODUCTION

In the Multics Operating System, segments are composed of 1024-word contiguous blocks of data called pages. At a given time, any number of pages of a segment may be located in core memory, but since that memory is limited in size, the hardware blocks of 1024-word core registers must be multiplexed among the many pages of data and procedures which may be referenced. It is the purpose of this chapter to detail the structure of the mechanism which accomplishes (block or) page multiplexing in Multics.

The page multiplexing strategy is similar in broad outline to the page table multiplexing performed by the segment fault handling module (see Chapter 5). However, there are differences in detail which arise in great part due to the 645 hardware used in page fault handling. Page fault handling is closely bound to the various registers and logic functions which the 645 processor can perform; indeed the major purpose of the paging modules is to create the proper environment for hardware access to pages. This access is made through several registers, but the one which uniquely concerns page multiplexing is the page table word (or PTW). This 36-bit register, located in the page table for a segment, contains all of the information used by the 645 processor to deal with a page. The proper maintenance of a PTW is page multiplexing's most basic job.

As a vehicle for carrying the description of page multiplexing, there is a convenient set of machine configurations whose physical capabilities obviate the need for various parts of the page multiplexing function. Considered in order of increasing likelihood they are:

1) Infinite core storage and no secondary storage,
2) Infinite core storage, with secondary storage, and
3) Finite core storage with secondary storage.

Although no multiplexing is required in the first two cases, we shall describe the Multics page fault handling strategy as it would be performed on each of these three configurations, since in this way the real strategy can be described incrementally.

Before discussing the handling of page faults in detail, we shall describe briefly the environment which allows a page fault to occur. It is important to remember that a page fault cannot occur if there is no page table for the segment in question - for a page fault is only generated by hardware reference to a page table. The table is provided for a segment by the segment fault handler when it activates the segment. Also at this time, the Active Segment Table Entry for the segment is initialized by the segment fault handler with all the information needed by the page fault handler. Finally, the page table is set with page faults for each page of the segment, so that the page fault handler will be invoked upon first reference to each page. These actions prepare the environment for page fault handling.

2. PAGE FAULT HANDLING ON A MACHINE WITH INFINITE CORE AND NO SECONDARY STORAGE

When a process attempts to reference a page whose PTW has a fault set, the paging modules are invoked to remove the fault and insert the proper core location of the page into the PTW. The first action, upon receiving notification of the fault, is to locate the page being sought. From the PTW address, the address of the Active Segment Table Entry (ASTE) for the segment can be found. The ASTE contains the segment map (the list of page locations) which yields the actual address of the page (as described in Chapter 7). The segment map is actually split between the ASTE and the page table; however, we will ignore this complication for the moment.

Since we are now assuming a configuration involving only core, the address must be that of a 1024-word block of core. Hence, the paging module need only insert the proper core address into the PTW and fill in the page fault field of the PTW to prevent further page faults on this page. Thereafter, references to the page through the PTW would proceed by hardware without interruption.

The repairing of a page fault in a PTW need be done only once in the environment we are assuming, because all Segment Descriptor Words (SDW's) for a segment point to the same page table and, therefore, to the same PTW for each page. If, however, a segment were deactivated (its page table destroyed), then before further references could be made to the segment, the page faults set by segment activation would have to be satisfied in the way described above. Page locations being constant assures us that the information stored in a PTW is valid as long as the page table is.

3. THE ADDITION OF SECONDARY STORAGE

If we introduce secondary storage into our machine configuration, we add two problems to the page fault handling mechanism. Since the location of a page can now be outside of core, there is a need to transport that page from its resident device; and also, we must find an appropriate core block into which to put it. The page fault path becomes slightly longer and necessitates referencing a new data base - the core map free list. For, in order to pick an appropriate block of core for the page, we must avoid those blocks currently in use. Since we are postulating infinite core, we need not be concerned with depleting the free page supply. The functions required in this configuration are:

1) Receive the fault, go from the PTW to the ASTE to get the segment map and determine the secondary storage location of the page.

2) Access the core map free list to obtain a 1024-word block and delete it from the list, and

3) call a Device Interface Module (DIM) to retrieve the page and deposit it in the newly acquired block of storage.

The DIM's functions in transporting pages are to queue requests for pages, to make the device perform as efficiently as possible in satisfying the requests, to monitor the device operation, and to notify the page fault handler when input has been completed.

The notification function is performed in such a way as to allow the process which took the page fault to wait for the page without wasting processor time or making unnecessary memory accesses. The page fault handler in the faulting process, after calling the DIM to initiate page transportation, calls the System Traffic Controller to wait for the page. At some later time, when the page has been imported, the Traffic Controller will be called to inform the waiting process that it may continue its computation.

4. RESTRICTION IN CORE SIZE

When we restrict our configuration to have a finite amount of core, we reach the true Multics case where multiplexing is necessary. In addition to the functions previously described, the page multiplexor must also be responsible for finding a free block of core when all blocks are being used. This condition requires a selection algorithm for removing pages from core to secondary storage; and this algorithm requires a new data base - the core map used block list.

The two lists of core blocks - free and used - are implemented by means of an entire core map. The core map consists of one core map entry (CME) for each 1024-word block of core. Each entry serviced by the page fault handler is threaded into one of the two queues - free or used - depending upon its current status, but the entry and its physical block remain associated throughout any change in status. The free list is singly threaded, since only its first element is ever used, but the used list is circularly threaded to allow continuous searching.

The actual information contained in a CME can be deduced from the part it plays in the removal algorithm. Clearly it must allow one to obtain the absolute core location of the beginning of the 1024-word block. But a pointer to the PTW for the page currently residing in the block is also necessary if the block is being used, since the removal of the page means that access to it should be inhibited by setting a page fault in the PTW which controls it.

The sequence of actions for handling a page fault in the limited core environment is:

1) As before, get the fault, find the device address from the segment map in the ASTE.

2) Access the core map free list to obtain a free block: if successful, place a pointer to the PTW in the CME selected, unthread the CME from the free list, flag it as being used for I/O and thread it into the used list. (Continue at Step 6).

3) If the free list is empty, perform the replenishment algorithm to find a block in which the referenced page can be put.

   The replenishment algorithm is driven by a bit in the PTW called the "Page has been used" or PHU bit. This bit is set by the 645 hardware whenever the page is accessed. When a page must be removed, the used block list of the core map is accessed and the entries are examined in the order of their threading, starting where the last search stopped. Each entry is examined to determine whether the PHU bit has been turned on. Each page's PHU bit is turned on by the software when the page is brought to core, but after each examination during the removal algorithm the page fault handler turns it off. Therefore, the effective criterion for removal is whether the page has been used since it was last examined for removal. (For further information about the philosophy of this algorithm, see "A paging experiment with the Multics System", F. J. Corbato, Multics Repository Document M0104.)

4) Having found a candidate for removal, set a page fault in the PTW, determine the device address from the ASTE and call the DIM to transport the page to secondary storage.

In order to avoid unnecessary page transportation, the 645 hardware maintains a "page has been modified" bit in each PTW which is turned on only if the page has been written into. Unless this bit is on, the page need not be returned to secondary storage.

5) In contrast to the page important strategy, do not wait for this page to be moved, but continue to select candidates for removal and call the DIM to transport them until a block appears on the free list. This block will have been placed on the free list by the DIM when a page has been completely transported.

6) Using the device address developed in step 1), call the DIM to import the desired page into the newly acquired block of core.

7) Call the Traffic Controller to wait for the page to arrive.

8) Since the PTW's page fault switch and CME's I/O busy switch are reset by the process which called the Traffic Controller to awaken the waiting process, the page fault has been entirely repaired and the page fault handler may return.

Figure 1 is a gross flow chart of the page fault handling strategy as described in Section 4. Special points of interest are:

1) Any call to the DIM, whether for reading or writing, causes all transactions to be observed and the completed ones to be "posted". The posting process for a write operation consists of threading the CME out of the used list and into the free list. For a read, posting requires that the page fault switch be reset and that any processes which might be waiting for the page to be imported be informed of its arrival (through the Traffic Controller).

2) The call to the Traffic Controller to wait for a page to be read is only made for reasons of efficiency and plays no logical part in the page fault handling strategy.

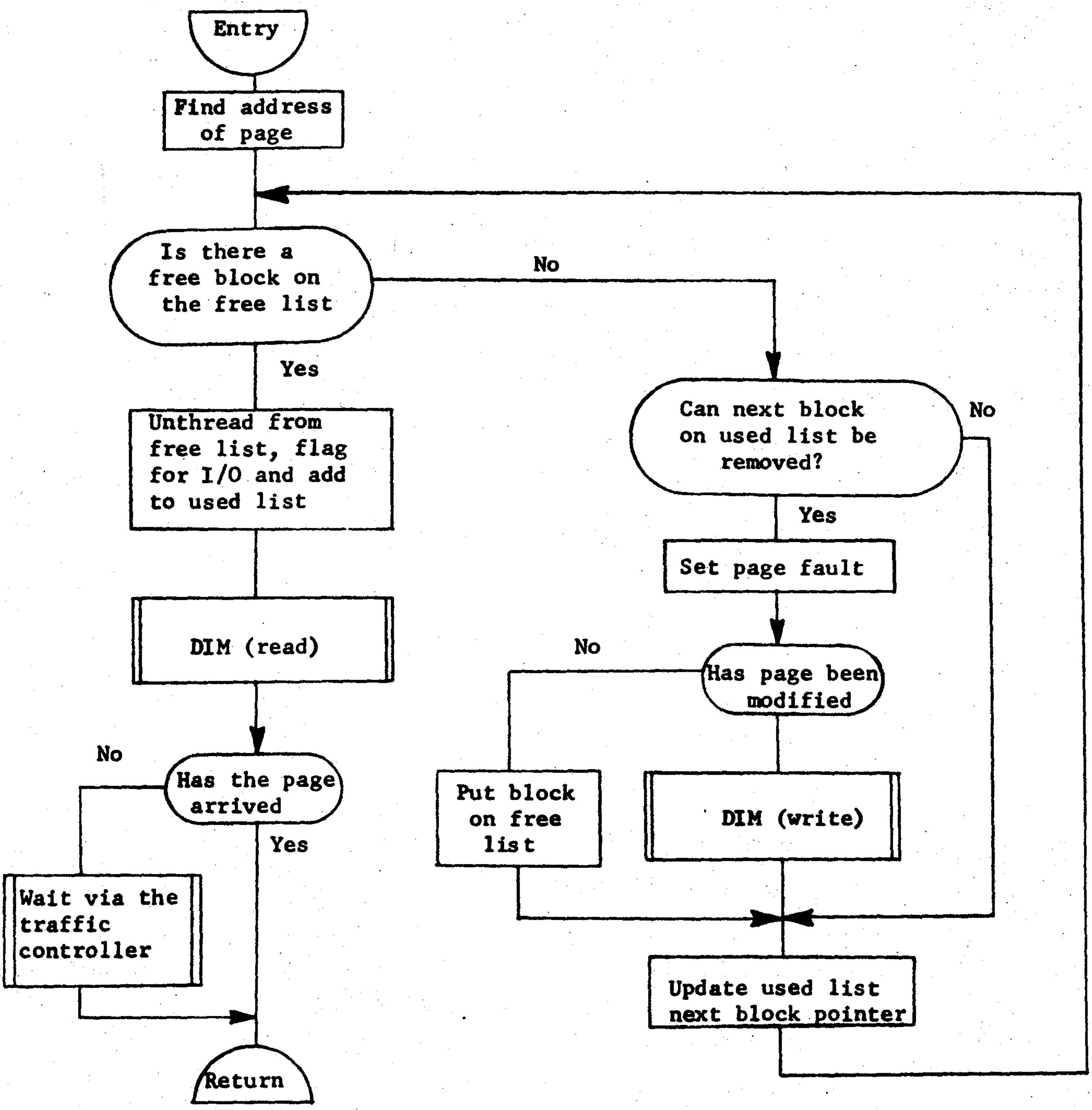


Figure 1. The Page Fault Handler

5. RECAPITULATION OF PAGE FAULT DATA BASE USE

Having followed the basic paths through the page fault handler, we have seen all of the data bases used by page multiplexing. However, not all of the items in these data bases have been specified, only those portions significant to a general understanding of page fault handling. This section describes all of the data items referenced by the page multiplexor and indicates their uses. (See also Figures 2 and 3).

The most basic page multiplexing data base for a segment is the page table, which contains the page table words for each page. Page tables in Multics are allocated at system initialization time in a permanently core-resident system-wide segment, the System Segment Table (see Figure 3). Hence, references to a page table are made through the normal segment addressing mechanism - although no page faults are ever taken on such references. Each PTW has six types of data used by the page fault handler. These types can be divided further into hardware referenced data and solely software referenced. Both types are important to page multiplexing.

The set of hardware referenced data in a PTW consists of three items - the page fault switch, the core address field and the page has been modified/used bits. Whenever the core address is not meaningful, the page fault switch is set to inhibit processor attempts to access through that address. Since the address field is not used when the page fault switch is on, the page multiplexor makes use of the storage thus provided by placing part of the device address in that field. Logically, the entire address can be considered to be in the Active Segment Table Entry as mentioned earlier, but to save storage space, the "segment map entry" for a page is stored in the core address field of its PTW when the page is not in core. The PHM and PHU bits, set by the hardware and reset by the software when appropriate, have already been described.

The data in the PTW which is only referenced by software consists of two flags used by the I/O handlers and a bit which designates "wired down" status. The latter is interpreted by the page fault handler during the page removal algorithm to mean that this page may not be removed. The two former flags are: one to signify that there is I/O pending for this page (and whether read or write) and another which is set if an I/O error is encountered while transporting the page.

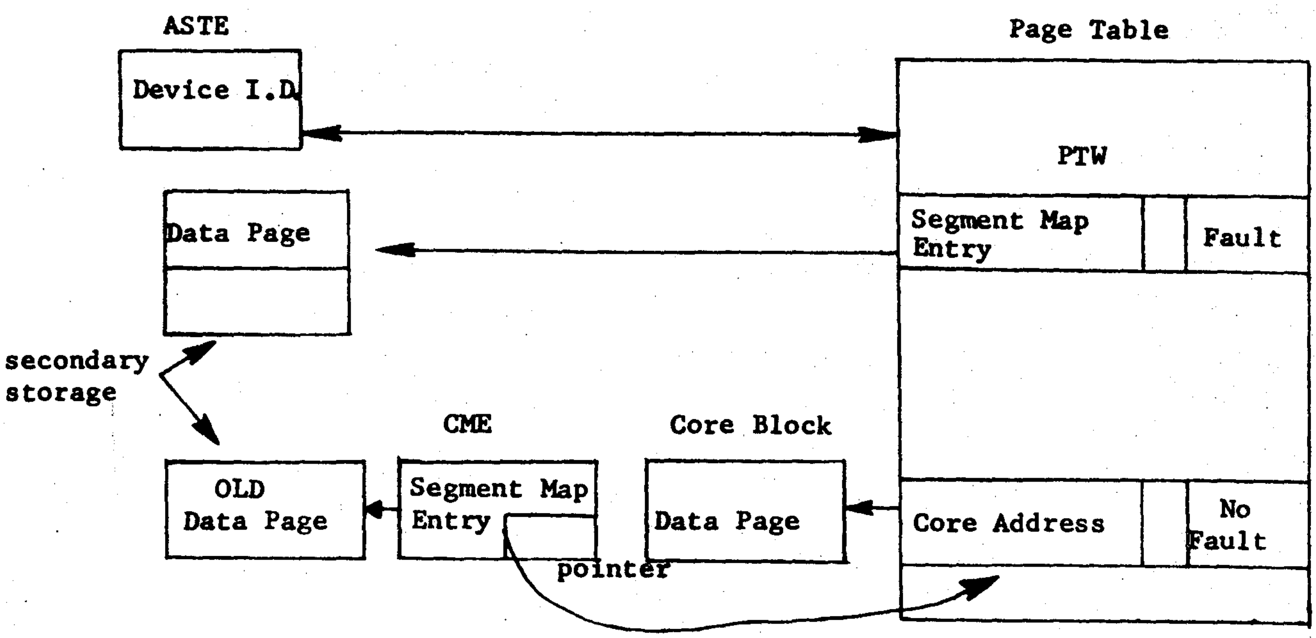


Figure 2. Data Base Inter-relationships

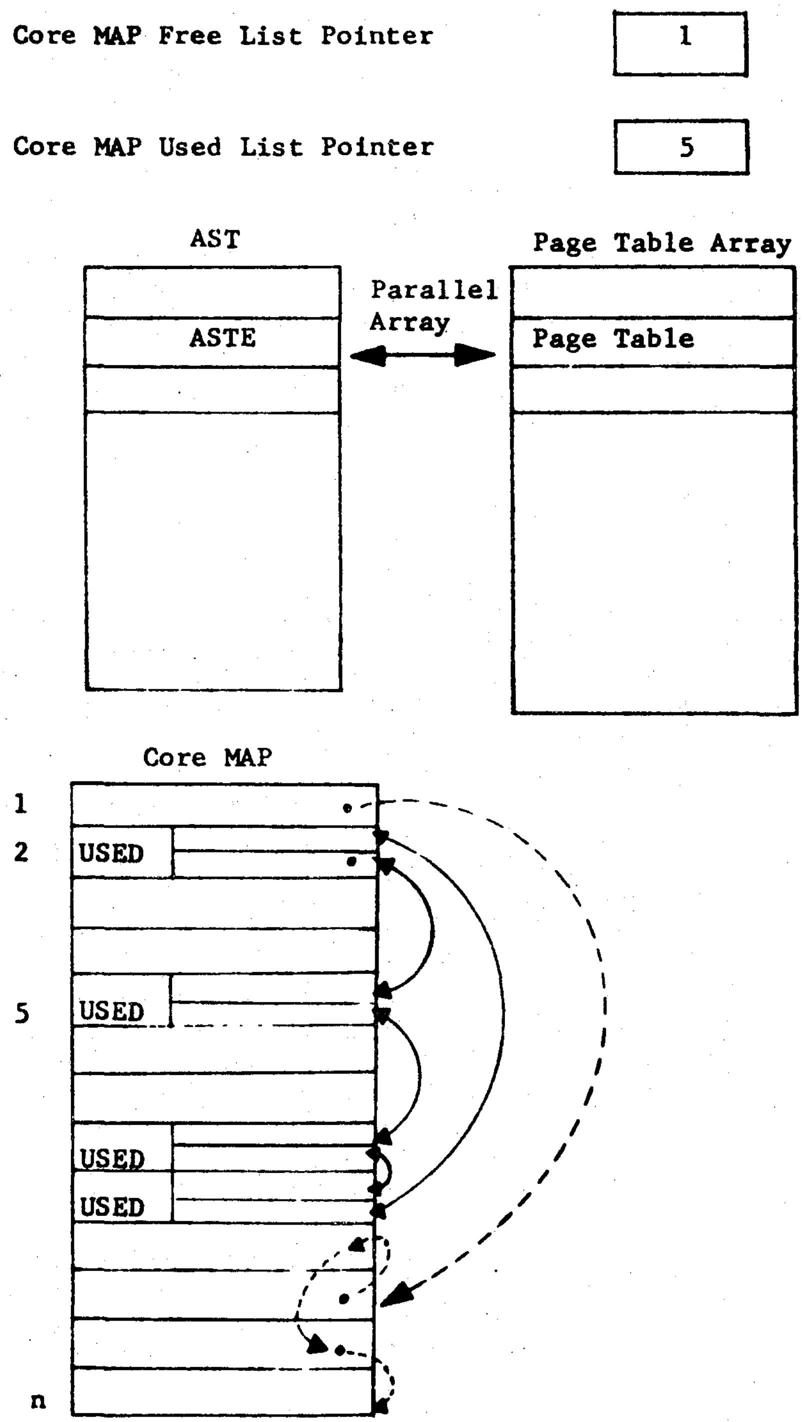
Notes for Figure 2.

A. To bring in a page from secondary storage, start with the PTW which took the fault. Go to the ASTE to get the Device I.D. Use the core map free list to get a suitable CME. Fill in the pointer to the PTW and transfer the file map to the CME, then read in the page and remove the fault.

B. To remove a page from core, start with the CME picked from the core map used list. Go to the PTW by the pointer and set a fault; thence to the ASTE for the Device I.D. to complete the secondary address. Move the file map back to the PTW. If the page has been modified, write it out.

NOTE: It may seem that Multics has two copies of a data page when the data is in core. Logically there is only one and we could easily free the storage used by the old page each time it was read in. There are three reasons of efficiency why we do not do so.

1) Assigning and reassigning secondary storage blocks takes processor time.
2) If the page to be removed from core has not been modified and we have retained the old copy, we do not need to write it out.
3) If the system were to crash, losing core but not secondary storage contents, we would still have a (possibly obsolete) copy of the data.



The n^{th} CME refers to n^{th} 1024-word block of core.

Figure 3. The System Segment Table

A fourth item of information, the address of the segment's ASTE, although not contained in the PTW, is implicit in its address; the ASTE's (which are fixed in length) are stored in an array parallel to the page tables, thereby yielding the ASTE address from the page table (or PTW) address. This information is necessary to provide the rest of the device address for a page when reading it in from secondary storage.

Another data base important to page fault handling is the Active Segment Table Entry (ASTE) for a segment. The ASTE contains more data than is used by the page multiplexor, but that portion which is used consists of the device I.D., the page fault count, a "no page fault" switch, the number of pages in core for the segment and the current segment length, in pages. Of these, only the device I.D. (see Chapter 7) is crucial to page fault handling - the others are measurements for tuning purposes and aids to Segment Activation and Deactivation which can be best provided by the page multiplexor. Additional comment is made on these items in Section 6 of this paper.

The third, and last data base used by the page fault handler is the core map. The core map is also allocated in the System Segment Table whose header contains pointers to the head of the free list and to the next entry to be examined on the used list. Each core map entry (CME) has two threading pointers: the entries in the free list use only the "forward" pointer, since entries are removed from the top and added there, while the used list employs both "forward" and "backward" pointers to allow insertion of an entry between any two other entries. A CME on the free list has no other useful information contained in it. The core location of the block controlled by a CME is implicit in its position within the core map, since as many CME's are allocated as there are 1024-word blocks of core. We note that not all CME's are put on the free or used threads - only those blocks to be serviced by the page fault handler. All permanently core-resident pages of core are represented by CME's which are not pointed to by entries on one of the threads. In this way, the removal algorithm need not explicitly check entries which could never be removed.

Two other items are kept in the CME for a page only if the CME is on the used list. First, the segment map entry for the page, which was kept in the PTW while the page resided in secondary storage, is transferred to the CME before the core address is inserted into the PTW. Clearly the core address could be kept in this space in the CME when it was threaded on the free list if the information were not implicitly available. Second, a pointer to the PTW is maintained to permit the segment map entry to be replaced and the page fault switch to be reset when the page is removed from core.

We are now in a position to understand the initial requirements of the page fault handler in order that it be able to function. First, the core map entries for multiplexible blocks must all be threaded onto the free list. Then, for each segment which is to be referenced, each page table word must be filled in with a segment map entry and page fault switch setting; and also the Active Segment Table Entry must be initialized in its Device I.D. and other entries. This work would allow a page fault to be serviced for non-page-fault-handling procedures.

But what of the page multiplexor itself? May it use the same page fault mechanism which it provides? Not entirely. While selected parts of the data and procedures of the multiplexor could be transportable, at least one path through the multiplexor must be guaranteed page-fault free to prevent infinite recursion. In Multics, the choice has been made to prevent any page faults whatever from occurring during the handling of a page fault. To this end, all of the page fault handling procedures are permanently core-resident, and for this reason the ASTE is used by the page multiplexor. For although all of the necessary information about the pages of a segment can be found in the branch for that segment, branches are not wired down and are, therefore, susceptible to page faults. Hence, that part of the information kept in the branch which must be referenced by the page fault handler is transferred to the ASTE at segment activation time, to keep it in a permanently core-resident data base. This choice permits Multics to follow a more predictable and shorter path while handling page faults.

6. ADDITIONAL REMARKS

Several important functions of the page multiplexor have been excluded from the previous discussion since they are not necessary to allow page multiplexing. Some of these functions receive additional coverage in chapters written to explicate the areas in which they are used, but their appearance in this paper is germaine to a better understanding of the page fault handler's importance in Multics.

An especially complex area in any multiplexed computing system is that of synchronization of processes. Empirical evidence has led Multics to the path of least complexity where possible. An example is the synchronization of multiple processes, all of which may desire the handling of a page fault at the same time. To prevent interference between the various processes in handling common data, only one process is allowed to execute in the page multiplexor at a time. Other processes wishing to deal with a page fault are forced to wait their turn.

It is also necessary to synchronize the physical devices which transport pages and the processes which have requested the transportation. As we have seen, there is no problem of synchrony when writing pages. Any process which needs to have a write operation completed in order to continue its computation simply loops on the core map free list and calls to the DIM until a block appears in the free list. When reading pages, synchrony is established through communication with the Traffic Controller - both to wait and to notify. The only remaining problem is to ensure that the DIM is called subsequent to the completion of every read so that notification can be performed.

The DIM is normally called by the next process to take a page fault. But if the DIM is not called in a sufficiently long time (this could happen if each process were waiting for a page!), the secondary storage devices cause interrupts which are directed to a special process whose sole purpose is to wait for these interrupts and call the DIM in response to them (see Chapter 8).

Another function performed by the page multiplexor is the assignment of blocks of secondary storage (see Chapter 7). No secondary storage is assigned to a page until it has been referenced. A special device address (the "null" address) is assigned to all pages of a segment which have never been referenced. When such an address is encountered by the page fault handler, it creates a page of zeroes in core rather than reading in data. Ideally, then, the page need not be written out until the page-has-been-modified (PHM) bit has been turned on by the hardware. In fact, the Multics page fault handler causes the assignment of a secondary storage address at first reference and sets the PHM bit to force write-out of the page. This sometimes results in storing a page of zeroes in secondary storage but eliminates a check for "null" addresses in the page removal path.

Actual device storage handling is incorporated into a separate module whose algorithm for assigning storage on a particular device can be easily changed to accommodate any system discipline (such as directory segments on Drum and non-directory segments on Disk).

The movement of data from one device to another is also accomplished in the page multiplexor. Chapter 7 describes the function in detail, but the basic mechanism used is an additional item in the ASTE for a segment which can specify a "move device I.D." and a bit in the segment map entry which specifies whether or not the page (which must be in core) has already been moved. Using these items, a segment can be moved by setting the move device to I.D. when activating the segment. Then, whenever a page is brought to and subsequently removed from core, it is rewritten onto the new device.

The final functional area incorporated into the page multiplexing modules is that of services to the segment activation and deactivation module. A special set of entry-points allows individual page manipulation on demand. Specifically, the functions are:

1) To <u>read</u> or <u>write</u> a page from/to secondary storage.

2) To "wire" or "unwire" a page by setting the "wired down" bit in the PTW which allows a page to be skipped by the removal algorithm (this function is

used only for temporary wiring). Pages are "wired" (made permanently core-resident) by leaving their CME's off the core-map used list.

3) To <u>truncate</u> a segment by destroying all pages beyond a certain page and returning their secondary storage to the free pool.

4) To <u>cleanup</u> all traces of a segment in preparation for its deactivation. This function consists of exporting the entire segment (removing all of its pages) and waiting until the pages have all been exported to their resident secondary storage.

7. THE HISTORY OF THE MULTICS PAGE MULTIPLEXOR

The entire Multics file system has gone through two incarnations. The original version carried the Multics penchant for elaborate original design to some lengths and was successfully implemented. Its performance and amenability to debugging left something to be desired. Therefore, this second attempt was made, using the knowledge gained from the first to avoid areas of difficult implementation and slow execution.

There were two principal changes respecting paging. First, a single page size of 1024-words was chosen, replacing the previous strategy in which pages of two sizes, 1024-words and 64-words, were allowed. This simplification resulted in the elimination of elegant but time-consuming algorithms for page removal and for "change-making" and coalescing free blocks in core and in secondary storage. Second, a single segment size of 64 pages (implying a single page table size of 64 words) was chosen, replacing the previous strategy in which segments could vary in size from 64 to 256 pages (always in units of 64 pages). This simplification resulted in a greatly simplified Page Table-Active Segment Table Entry arrangement to the benefit of all the modules involved: page control, segment control, core control. As a result of these changes, the present implementation has avoided several lengthy computations in the most frequently used path in page fault handling, achieving a great advantage in average execution time over the former implementation.

Chapter 7

SECONDARY STORAGE MANAGEMENT

1. INTRODUCTION

Secondary storage management cuts across many parts of the Multics virtual memory system. In this chapter, we shall try to minimize repetition by discussing only those points which have not been discussed elsewhere.

We shall discuss the assignment of a segment to a secondary storage device and the assignment to its pages of blocks of secondary storage. We shall also discuss "moving" a segment from one secondary storage device to another, i.e., changing a segment's assigned device.

2. ORGANIZATION OF SECONDARY STORAGE

The physical devices used for storing information in Multics--core, disks, drums--are divided into 1024-word "blocks" corresponding to the division of segments into 1024-word "pages". Addresses of blocks in secondary storage are given as pairs:

block address = (device identifier, block number)

For the most part, the "device identifier" specifies a particular physical device. It is possible, however, that one device identifier specifies part of a large device or a collection of small devices. We should, therefore, use the phrase "logical device identifier."

Each (logical) device of secondary storage has an associated "Device Map" which records which of its blocks are assigned to pages of segments and which are free. The "Device Map" contains one bit per block of the device. This bit is set to "1" to indicate that the block is free and to "0" to indicate that the block is assigned to a page.

To assign a block on a device to a page, it suffices to search the appropriate Device Map for a bit set to "1", note the corresponding block number, and reset the "1" to "0". To free a block (when a page is destroyed, for instance) it suffices to reset the corresponding bit in the Device Map from "0" to "1".

3. SECONDARY STORAGE OF SEGMENTS AND PAGES

3.1. Strategy for Secondary Storage of Segments and Pages

3.1.1. Segment Assignment. Devices of secondary storage are not equivalent. Due to differences of latency and transmission timings and to differences in the accessing code, some devices prove to be faster than others. For example, in the present Multics configuration, the drum is faster than the disk. Because of this non-uniformity, each segment is assigned to a single secondary device: segments expected to be used often are assigned to fast devices, segments expected to be used more rarely are relegated to slower devices. (When we say that a segment is assigned to a device, we mean that the pages of the segment are to be stored in blocks of that device.)

A segment is assigned to a device of secondary storage when the segment is created. The algorithm by which segments are initially assigned to devices of secondary storage is called the "Multi-Level Storage Algorithm". The phrase "Multi-Level" emphasizes the differences in device characteristics. A discussion of the Multi-Level Storage Algorithm is beyond the scope of this paper. We shall examine only the mechanisms used to execute this algorithm's decisions.

3.1.2. Page Assignment. When a segment is created, its 64 pages are also, in a sense, created. But before a page is referenced for the first time, it cannot contain any information.

Although a few segments may ultimately contain 64 information-filled pages, many segments never contain more than a few such pages. It would be wasteful to tie up blocks of secondary storage for pages that contain no information and may never be referenced. Therefore, pages are assigned blocks in secondary storage only after they have been referenced.

3.2. Data Relating to Secondary Storage of Segments and Pages

3.2.1. Segment Map. We may now specify the "Segment Map", that attribute of a segment which tells where the pages of the segment are stored in secondary storage. The Segment Map consists of:

- device identifier
- 64 block numbers

3.2.2. The "Null" Block. A special block number, called the "null" block number, is used to indicate that a page has not been assigned a block of secondary storage. We often say of such a page that it is assigned the "the null block". The null block may be regarded as a page of zeros.

3.3. Procedural Implications of Secondary Storage Strategy

3.3.1. Creating a Segment. When a segment is created, the Multi-Level Storage Algorithm is used to assign the segment to a device of secondary storage. The segment's 64 pages, which have not been referenced, are all assigned the "null" block. The device identifier and the 64 "null" block numbers are all recorded in the segment's Segment Map.

3.3.2. Bringing a Page to Core. When a page is referenced for the first time, the Page Fault Handler (PFH) is asked to "bring to core" a page which is "stored" in the "null" block. The PFH handles such a request by:

- assigning a block to the page from the segment's assigned device.
- zeroing out the block in core which is to contain the new page.
- setting the "page has been modified" switch in the page's PTW to make sure that the page will ultimately be moved to its newly assigned block.

3.3.3. Removing a Page from Core. When the PFH wishes to use a block of core presently occupied by a page, it inspects that page's "page has been modified" switch. If the page has been modified, then it must be written into its assigned block in secondary storage. If the page has not been modified, then it may be overwritten directly since it is equivalent to the information in the assigned block of secondary storage.

4. MOVING A SEGMENT FROM ONE SECONDARY STORAGE DEVICE TO ANOTHER

4.1. Strategy for Moving a Segment

It is occasionally necessary to "move" a segment from one secondary storage device to another. A "move" is necessary, for instance, if a secondary storage device becomes full or if a segment's usage changes substantially. The decision to re-assign a segment to a new device (to "move" a segment) is made by the same Multi-Level Storage Algorithm which assigned the segment to a device at segment creation time.

4.2. The Segment Map Revised to Permit "Moves"

The Segment Map must be revised if it is to be possible to move a segment from one device to another. The Segment Map must indicate not only the device to which the segment is assigned (in the case of a move, the segment is assigned to the new device) but also the device to which the segment was assigned. During the move, information must also be stored to show to which of these two devices the segment's 64 pages are assigned. Last, the Segment Map must show whether or not a "move" is in process.

The revised Segment Map has the form:

- device identifier (or, in case of a move, "old device identifier")
- new device identifier (non-zero only if a move is in progress; thus acts also as a "move in progress" switch)
- 64 block numbers
- 64 "moved" switches (a page's "moved" switch shows to which of the two devices the pages are assigned; the "moved" switch is meaningful only when a "move" is in progress)

4.3. Procedural Implications of "Moves"

4.3.1. Bringing a Page to Core. When a page is brought to core, it must be brought from its presently assigned block in secondary storage. The Page Fault Handler (PFH), by inspecting the "new device identifier", "old device identifier", and the page's "moved" switch, can determine the device to which the page is assigned. The page's secondary storage address then consists of the device identifier so calculated and the page's block number.

In the case of a "null" block assignment, the page referenced for the first time is assigned to a block of the new device.

4.3.2. Removing a Page from Core. When the PFH removes a page from core it must see that the page is removed to a block in the correct device. This means that:

(a) If the segment is being moved from one secondary storage device to another, and

(b) if the page's "moved" switch shows that the page has NOT been moved, then the PFH must

(c) release the block assigned to the page on the "old" device,

(d) assign a block to the page on the "new" device,

(e) record the new assignment in the Segment Map, setting the "moved" switch, and

(f) move the page from core to its newly assigned block in secondary storage.

4.3.3. Deactivating a Segment - Completing a Change of Devices. We know that when a segment is deactivated, the Segment Fault Handler calls a special entry of the paging module to force the segment's remaining pages out of core. Before it removes any pages from core, this procedure checks to see if the segment being deactivated is being moved from one device to another. If the segment is being moved, code is executed which brings to core those pages of the segment which are stored in blocks of the "old" device (that is, pages which have non-"null" block numbers and whose "moved" switches say "not moved".) When this has been done, the job of removing the segment's pages from core is performed. The pages are removed from core as described in the previous paragraph. Thus, at the end of the page removal, all of the segment's pages are necessarily assigned to blocks of the "new" device.

When the page removal procedure returns to the Segment Fault Handler, the latter updates the Segment Map to show the correct device identifier, a zero new device identifier, and all "moved" switches showing "not moved". With this, the move is complete; we see that deactivation completes a "move".

4.3.4. <u>Performing a "Move"</u>. We may now describe the procedure which the Multi-Level Storage Algorithm uses to move a segment from one device to another. The "new-device" identifier is placed in the Segment Map of the segment (in the branch if the segment is not active, in the ASTE if the segment is active). If the segment is not active, it is made active. Finally, the segment is deactivated by means of a call to a special entry of the Segment Fault Handler. By supplying a "new-device" and activating the segment, the "move" is initiated. By forcing the deactivation of the segment, the "move" is terminated, as described in the previous paragraph.

Chapter 8

DEVICE INTERFACE MODULES

1. INTRODUCTION

The responsibility of the Device Interface Module (DIM) respecting paging is (a) to initiate transfers of pages between blocks of core and blocks on secondary storage devices as requested by the Page Control Module, and (b) to notify the Page Control Module upon the completion of these transfers. In general, there is a considerable time lapse between the performance of these two functions.

Page Control uses the DIM by (a) initiating a transfer and then (b) waiting to be notified by the DIM of the completion of the transfer. The DIM must, therefore, perform its notification function with respect to a given transfer without being called by the process which requested that transfer.

Notification of completed transfers is usually performed by the DIM just after the latest transfer is initiated. The DIM inspects its data bases, determines which transfers (by whatever process requested) have been completed, and performs the necessary notifications. It then returns to its caller which may in turn wait for notification.

It may happen that all processes are waiting and, thus, that no process will (by taking a page fault) invoke the DIM. Therefore, as a precaution, the DIM arranges to have an interrupt sent to the processor by the secondary storage device sometime after it completes its last pending transfer. This interrupt will cause a special process to be wakened which will call the DIM and cause the required notifications to be performed. The DCW (see below) which causes this interrupt also disconnects the controller; we call it the "last" DCW.

2. GENERALITIES

2.1. DCW's

We are concerned with the I/O transactions of transferring a page from a block of core to a block of secondary storage (or vice versa) involving the GE-645 computer and its peripheral units. The Page Control Module "requests" such a transfer by a call to the DIM of the appropriate device. The DIM passes the request along to the device controller via a special word, called a DCW - Data Control Word. A DCW contains the following information of interest to us in this paper:

- op-code	read, write, or no-op (on some devices)
- core address	
- device address	
- disconnect bit	causes the device to disconnect after satisfying the request specified in the op-code
- interrupt bit	causes the controller to send an interrupt to the processor after the request has been satisfied

2.2. DCW Lists

Each device controller is driven by a list of DCW's which it runs through, consecutively, interpreting the DCW's, until it encounters a DCW with a disconnect bit set after satisfying which the controller disconnects itself and waits to be reconnected. The DCW lists are all regarded as circular in the sense that the controller accesses the DCW's consecutively, modulo some N. The DCW lists are finite in the sense that the DIM's always store a DCW with a disconnect bit somewhere in each DCW list. This DCW, whose interrupt bit is also set, is called the "last" DCW.

2.3. Status Queues

While the device controllers obtain their instructions from DCW lists, they record the status of requested transfers in special "status queues". A word in the status queue is associated with a DCW and written into by the controller after the transfer specified by the DCW is begun. The status word will show whether the action begun was completed, whether there was a parity error, etc.

2.4. Function of the DIM

The DIM for a particular device interfaces with the device through the two data bases discussed above, the DCW lists and the status queues. The DIM interfaces with the user (the Page Control Module, in our case) as follows:

- on being called, the DIM sets up appropriate DCW's,
- makes sure that the interrupt and disconnect bits are set in the proper DCW, and
- connects the controller if necessary.

After acting for the particular user, as above, the DIM

- inspects the status queue,
- "posts" all completed transfers in the associated Page Table Words,
- "notifies" the processes waiting for the completed transfers, and
- cleans up the SDW list and status queue as required.

2.5 Normal Operation

It is expected that enough page faults will occur that "requests" for page transfers will, in general, occur while each DCW list is non-empty. This means that the controller has not yet reached the "last" DCW with its disconnect and interrupt bits set. The DIM accordingly establishes new DCW's (as indicated above) and then advances the "last" DCW, that is, resets the disconnect and interrupt bits in the DCW in which they are presently set and sets them instead in the now appropriate DCW, further along on the circular DCW list. (We will discuss in detail below just which DCW is "last".) In this way, it is expected that the controller will run for a long time without reaching the "end" of the DCW list and will consequently remain connected and will not have to send interrupts. Since the purpose of the interrupts would be to force the invocation of the DIM to "notice" the completion of transfers, and since the DIM will be called quite often as part of paging, there is no need for interrupts. The interrupt associated with the "last" DCW takes care of the unlikely case that all processes are waiting for page I/O and that no more calls to the DIM from the "user" will occur. In this case, the interrupt will force a special process to be wakened which will call the DIM and so enable the noticing of completed transfers, their "posting", and the associated notification.

3. DRUM

3.1. Drum Configuration

The drum contains M*N blocks of 1024 words as shown in Figure 1. As the mth row of blocks comes under the read/write head, any of the N blocks in the row can become part of a transfer.

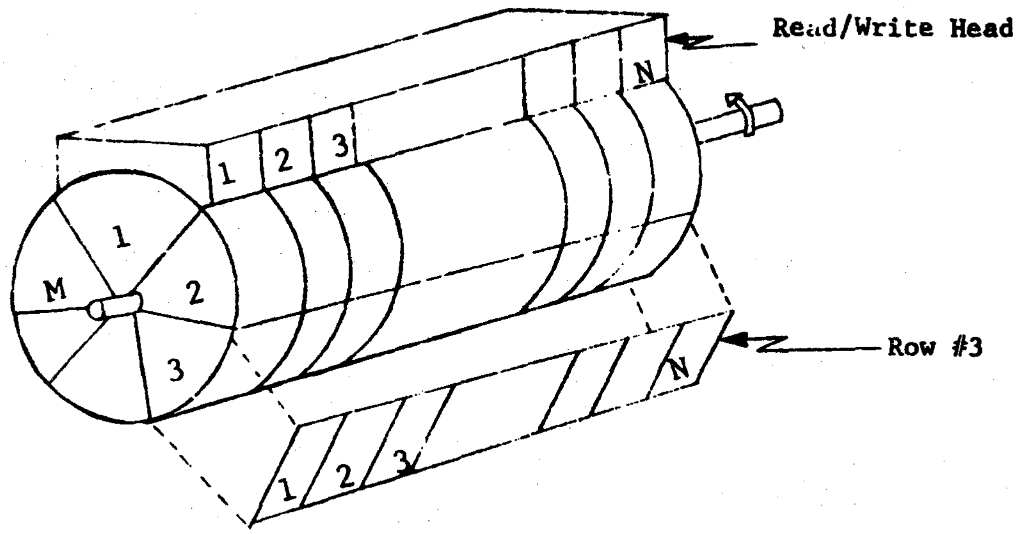


Figure 1. Drum Configuration

3.2. Drum's DCW List Configuration

The drum DIM maintains a circular DCW list for the drum of length L*M where L>>N. Each DCW contains the following information:

> (read, write, or no-op), core address, m, n, ("last" or "not-last")

where "last" means the disconnect and interrupt bits are set (in one DCW only), m is the row number, and n is the number of the block in the row. The drum's DCW list is initialized with all of the m's set, in order, so that the DCW list consists of L one-entry-per-row coverings of the drum. As the drum rotates and as the controller advances through the DCW list, the number of the drum's "presenting" row equals the row number of the DCW then pointed at by the controller. For this reason, the drum DCW may have a no-op op-code to specify that no transfer is to occur involving a block on the mth row at this time. See Figure 2. It should be emphasized that this L-fold "covering" of the drum by the DCW list and the consequent parallelism between the drum's physical position and the words of the DCW list is not required by the hardware but is a software construct (of some beauty.)

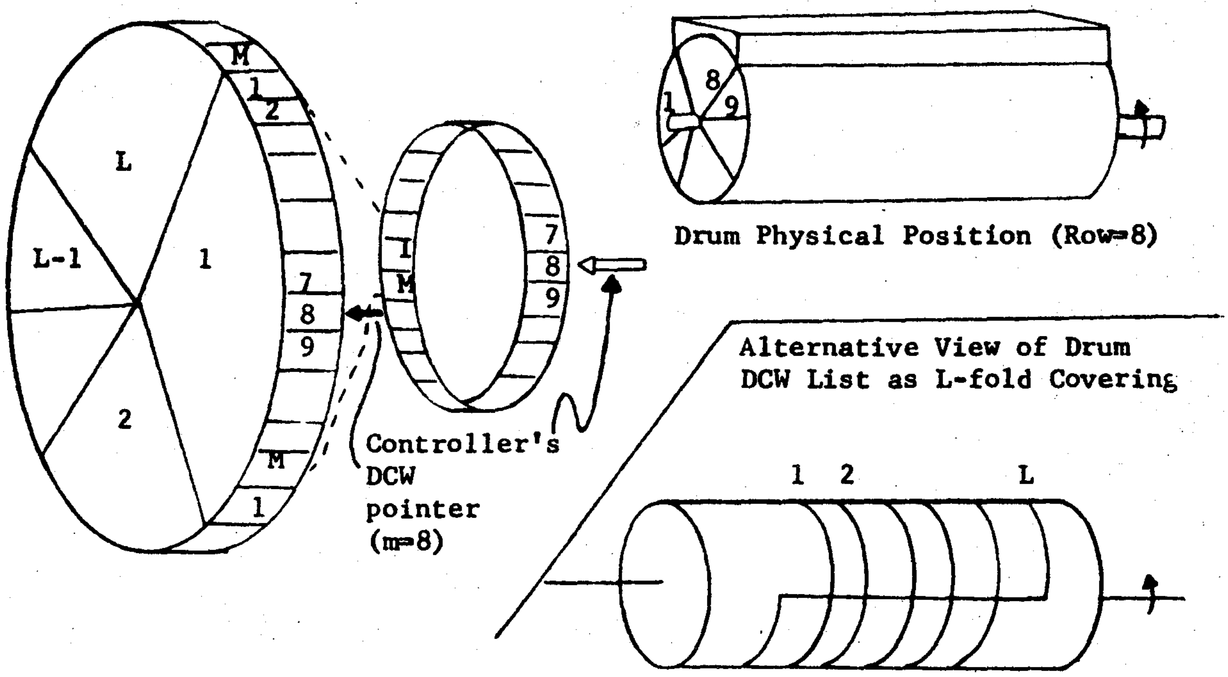


Figure 2. Drum DCW List Configuration

3.3. The "last" DCW

Whenever it is called, the drum DIM resets the "last" DCW to be the DCW last passed by the controller, that is, the DCW for which the drum controller has just completed the recording of final status. This guarantees that the drum controller will never generate an interrupt (and disconnect) until L revolutions of the drum after the last call to the drum DIM. The drum DIM maintains a pointer to the "last" DCW and, upon being called, erases the "last" information from the presently "last" DCW and writes it into the DCW just passed by (as explained above), resetting the "last" pointer.

3.4. DCW List Management for the Drum

When a transfer request is sent to the drum DIM, a pair (m,n) is sent as drum block address. The DIM examines the DCW list and finds that unique DCW in the L*M DCW's of the list which (a) will first be in the controller's path, (b) has a no-op op-code, and (c) has the given value of m in its row number slot. There are L DCW's with row number m and it is expected that at least one of them is no-op. (If all L of them are in real use, the DIM loops, waiting for one of them to come free.) When such a DCW is found, the DIM writes the appropriate value of n into the DCW and sets the appropriate op-code (read or write). The DIM then goes through the usual steps of setting the "last" position correctly, observing, posting, and notifying for completed transactions. One part of cleaning up after completed transactions is to reset to "no-op" the op-code of DCW's whose requests have been serviced.

It is expected that the drum controller will, in general, continuously run through the circular DCW list, the "last" DCW running along L revolutions behind it, with a band of non-null DCW's just in front of the current controller position in the list. See Figure 3.

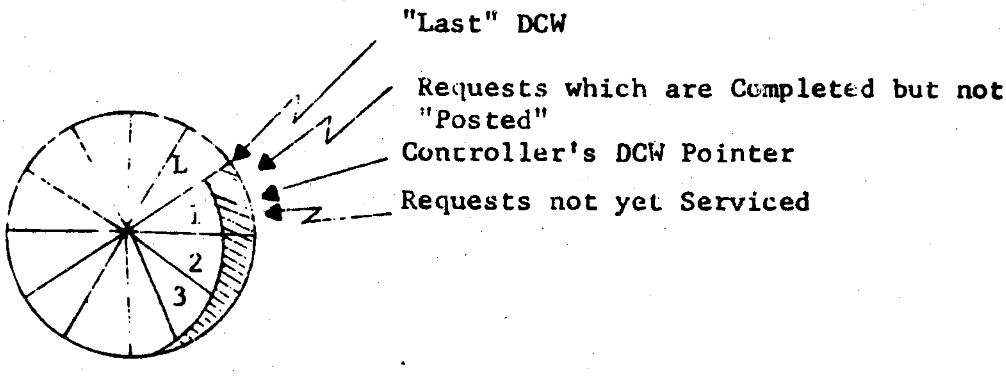


Figure 3. Density of Non-Null DCW's in Drum's DCW List

4. DISK

4.1. Disk Configuration

The "disk" consists of I platters each with one movable I/O head. The I/O head for any platter may be moved to any of J tracks. Each track has K blocks which come under the I/O head as the disk rotates. A disk DCW specified:

> (read or write), core address, i, j, k, ("last" or "not-last")

4.2. Disk DCW List Configuration

The disk's DCW list is circular only in the sense that its DCW's are accessed by the controller consecutively modulo the list length. DCW's are put on the list in the order received; there are no no-op DCW's, and the last DCW to be put on the list is always the "last" DCW in the sense that it contains the disconnect and interrupt bits.

4.3. Expected Operation

The Disk DCW list is ordered randomly in the sense that requests are put on the list as received and hence without regard for the position of the disk (k) or of the I arm positions j(i). This technique is chosen because of the large amount of processing that would otherwise have to be done to keep track of the arm positions, the value of k, and the list of unsatisfied DCW's; and because of the low probability that such processing would pay off.

No interrupt and disconnect will occur as long as there is at least one request in the queue. Since no reuse of old DCW's is made (as in the drum's case), there is no need, in cleaning up, to erase the contents of the DCW's of completed requests.

B. Access Control to the Multics Virtual Memory

CONTENTS

INTRODUCTION

An important trend in the design of large computer systems is the inclusion of hardware and software for the sharing of information, both procedure and data. Thus, the concept of pure, re-entrant procedure has lost its novelty and the sharing of data, as in multiuser information retrieval systems, has become commonplace. The introduction of sharing into large systems has, however, brought the difficult problem of access control into the realm of the computer system. The comparatively easy problem of protecting the supervisor in a batch environment has grown into the complex task of permitting the flexible sharing of information between system and user and between user and user.

The Multics access control system has been described in a number of places with a number of purposes. Graham[3] discusses the fundamental reasoning behind the choice of the Multics ringed access control system; Organick[4] discusses the details of the implementation and use of this system; and the Multics System Programmers' Manual goes into even greater detail on implementation. The purpose of the present paper is not to duplicate any of the excellent material already available but rather to highlight certain aspects of the Multics ringed access control system which are thought to be of particular interest to system programmers.

In this paper we shall develop the ides of the ringed access control system as an approximation of access control conditioned on the identity of the procedure in execution, as suggested by Evans and Leclerc[2]; we shall describe "ringed hardware" to support the ringed access control system; we shall show how this "ringed hardware" is simulated on the 645 processor; and we shall discuss at some length the software mechanisms which are implied by the concept of "ring".

This paper was written in conjunction with, and logically follows, another paper, The Multics Virtual Memory[1].

Chapter 1

ACCESS CONTROL PHILOSOPHY

In the Multics virtual memory, the segment is the unit of information to which access is controlled. In fact, the possibility of controlling access to shared information was a principal justification for designing a segmented memory system. In Multics, every segment is directly addressable and it is, therefore, necessary, upon each attempted memory access, for the accessing hardware to answer the question:

Shall this attempted access be permitted?

The answer to this question, with respect to a given segment, is obtained by interpreting a data base associated with the segment, the segment's "access control attributes". It is the purpose of this chapter to discuss the basis on which the hardware might go about answering this question, hence, to specify the content of a segment's access control attributes.

We feel that, at the least, a segment's access control attributes should indicate:

1. _who_ may access the segment; a segment may be accessible to a single user only or shared by a number of users.

2. _how_ each of these users may access the segment; distinct users may have distinct access rights.

3. _in what circumstances_ each user may exercise his access rights to a segment; a user's rights may be made to depend, in some way, on what he is doing.

User-Name. Let us look at these points in turn. To begin with, it is a process executing on a processor which attempts to access memory, not a user. For this reason, every process has associated with it the name ("user-name") of the user on whose behalf it is executing; all access rights of the process derive from the process' user-name.

We note that the security of an access control mechanism depending in this way on the user-name depends strongly on the technique by which a process is assigned a user-name.

A simple and perhaps sufficient technique for assigning user-names to processes is to require each user, when he "logs in", to specify a user-name and then give a secret password which validates his right to use the given user-name. All processes which subsequently act for him as a result of this "login" will then do so with the authority of the given, validated user-name.

Access-Mode. The basic types of memory access are READ, WRITE, and EXECUTE. We use the term "access-mode" to refer to any combination of these types including the null combination. It is clear that a process' access rights respecting a segment are at any moment characterizable in an access-mode.

If a process' access rights were to be independent of its activities, then a segment's access control attributes could be recorded in a list of user-name/access-mode pairs. A process' right to access a segment in a given way would then be determined by (a) whether the process' user-name appeared in the segment's list and (b) whether the given access type appeared in the corresponding access-mode. This system of access control is illustrated in Figure 1. The access control mechanism takes as arguments the process' user-name, the type of the attempted access, and the name of the target segment. It then searches the target segment's access control attributes list for the given user-name. If it is found, the corresponding access-mode is then searched for the given access type. The access is permitted only if the user-name and access type are found.

The system of access control just described is already quite powerful. It permits a user who has created a segment to grant himself READ-and WRITE-access to it, to store information in the segment, and then to give a number of other users READ-access to it. He and these others may now read the segment whereas he retains for himself the right to change it.

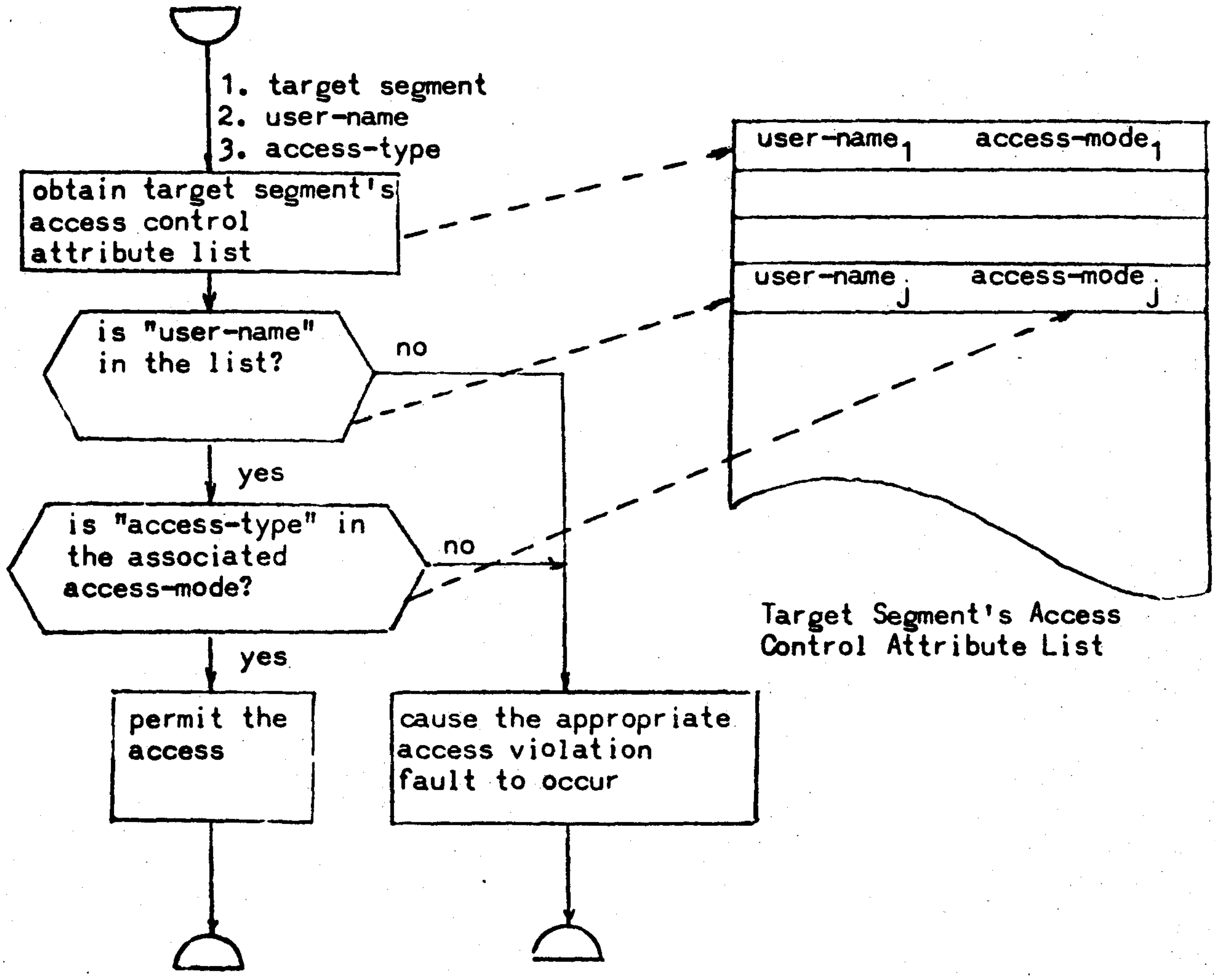


Figure 1. Illustrating an Access Control Mechanism Depending on User-name and Access-type

Circumstance-Dependence of Access Rights. There are two reasons why a process' access rights should somehow be made to depend on the process' current business. First, the problem of error may suggest that access, particularly WRITE-access, to a segment should be limited to debugged procedures or groups of procedures. Second, the problem of intentional misuse of a segment may suggest that semi-trusted users be forced to access the segment via procedures or groups of procedures specifically designed for their use.

As an example of the latter, consider a Management Information System with a data base including individual salary information. This data base would generally be "readable" by all users of the system; but the less privileged users would have to "read" the segment via procedures designed not to disclose individual salaries.

In the remainder of this chapter we shall take for granted the dependence of access rights on user-name and shall concentrate on finding a good and workable way to make a process' access rights to a segment circumstance-dependent.

1. ACCESS CONTROL BY PROCEDURE

The most obvious way to achieve circumstance dependence in access control would be to condition a process' access rights on the procedure by which the access is attempted. A segment's access control attributes would then be recorded, in effect, in a user-name versus procedure table whose entries are access-modes.

Figure 2 illustrates this system of access control. The access control mechanism takes as arguments the process' user-name, the type of attempted access, the name of the procedure in execution, and the name of the target segment. The target segment's access control attribute table is then searched for an entry corresponding to the given user-name and procedure. If the entry is found, the corresponding access-mode is then searched for the given access type. The access is permitted only if an entry with a suitable access-mode is found.

This type of control would permit access control as illustrated in the following example:

> user-1 "owns" data segment D and procedure P and gives user-2 WRITE-access to D only when executing P and EXECUTE-access to P only when executing in procedure P (and, of course, P itself).

This implies that user-2 can only write in D by calling P from P (to which user-2 has access from some source other than user-1).

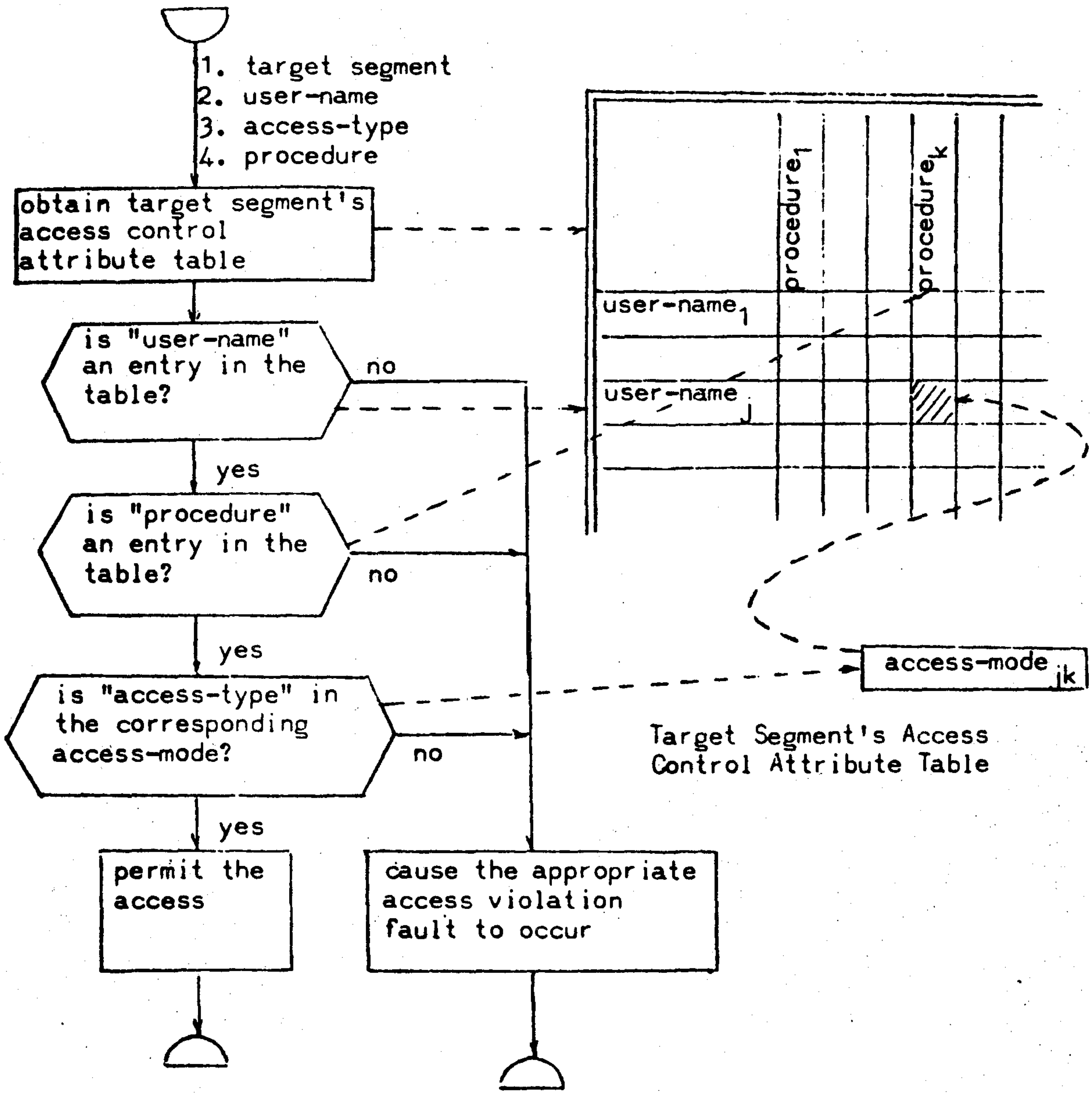


Figure 2. Illustrating an Access Control Mechanism Depending on User-name, Access-type, and Procedure

2. ACCESS CONTROL BY SET

The idea of conditioning access rights on the procedure-in-execution has been proposed by Evans and Leclerc[2] and is an idea that occurs to many system programmers at some point when they are struggling with difficult access control problems. We would recommend this technique were it not for some difficulties which render it infeasible. The principal hindrances to the conditioning of access rights on the procedure-in-execution are:

- no hardware presently exists which would permit this type of access control to be practiced in any but an interpretive mode

- too much effort and space must be expended in constructing and updating each segment's table of access control attributes

- too much must be foreseen: This technique requires knowledge of all of the uses to which each segment may legitimately be put.

A natural idea for approximating the procedure-in-execution strategy is based on grouping related procedure segments into "sets" and basing access rights to segments on the identity of the set to which the procedure attempting the access belongs.

There is no reason, by the way, to suppose that these sets of procedures would be disjoint; indeed, service procedures such as PL/1 runtime routines would probably be included in every set.

Access control by procedure-set appears to have two advantages over access control by procedure. First, each segment would have a somewhat smaller table of access control attributes, a practical system presumably having fewer sets than procedures in sets. Second, updating the per-segment access control attributes tables should be easier, since adding another procedure to a set would mean revising the definition of the set, not amending the access control attribute tables of a large number of segments.

Figure 3 illustrates access control based on set-in-execution. The form of a segment's access control attributes and the interpretation of these attributes by the access control mechanism are just as in access control by procedure, as described in Section 1 above, except that "set" replaces "procedure" wherever it occurs.

The concept of "sets" introduces a number of interesting problems. Given that each procedure is potentially an element of several sets, and stipulating that a change of set can only occur upon a change of procedure (i.e., upon a call or return), how shall the access control mechanism determine to which set to change (if any) upon each transition between procedures? How shall the composition (membership) of sets be initially defined and by what mechanism shall the composition of sets be changed? Shall the composition of sets be determined in a system-wide way or per-user or per-project? And so on.

We have introduced this concept of "sets" of procedures in order to make the definition of "rings" (see below) less abrupt and also to put the concept of "rings" in perspective.

3. ACCESS CONTROL BY RING

The implementation of an access control strategy based on sets, as described above, is judged infeasible due to the difficulty of defining sets, of unambiguously defining all transitions between sets, etc. It is useful to define a restricted theory which produces the more useful concept of "rings".

We use the term "rings" to refer to "sets" (as described above) to which access rights to all segments are assigned in such a way that the sets can unambiguously by ordered by increasing power or privilege. Precisely, we say that a collection of <u>sets</u> is a collection of <u>rings</u> if the sets can be numbered 0, 1, 2, ... in such a way that the possession by a particular user of an access right to a segment in set k implies the possession by that user of that segment for all sets j, j<k.

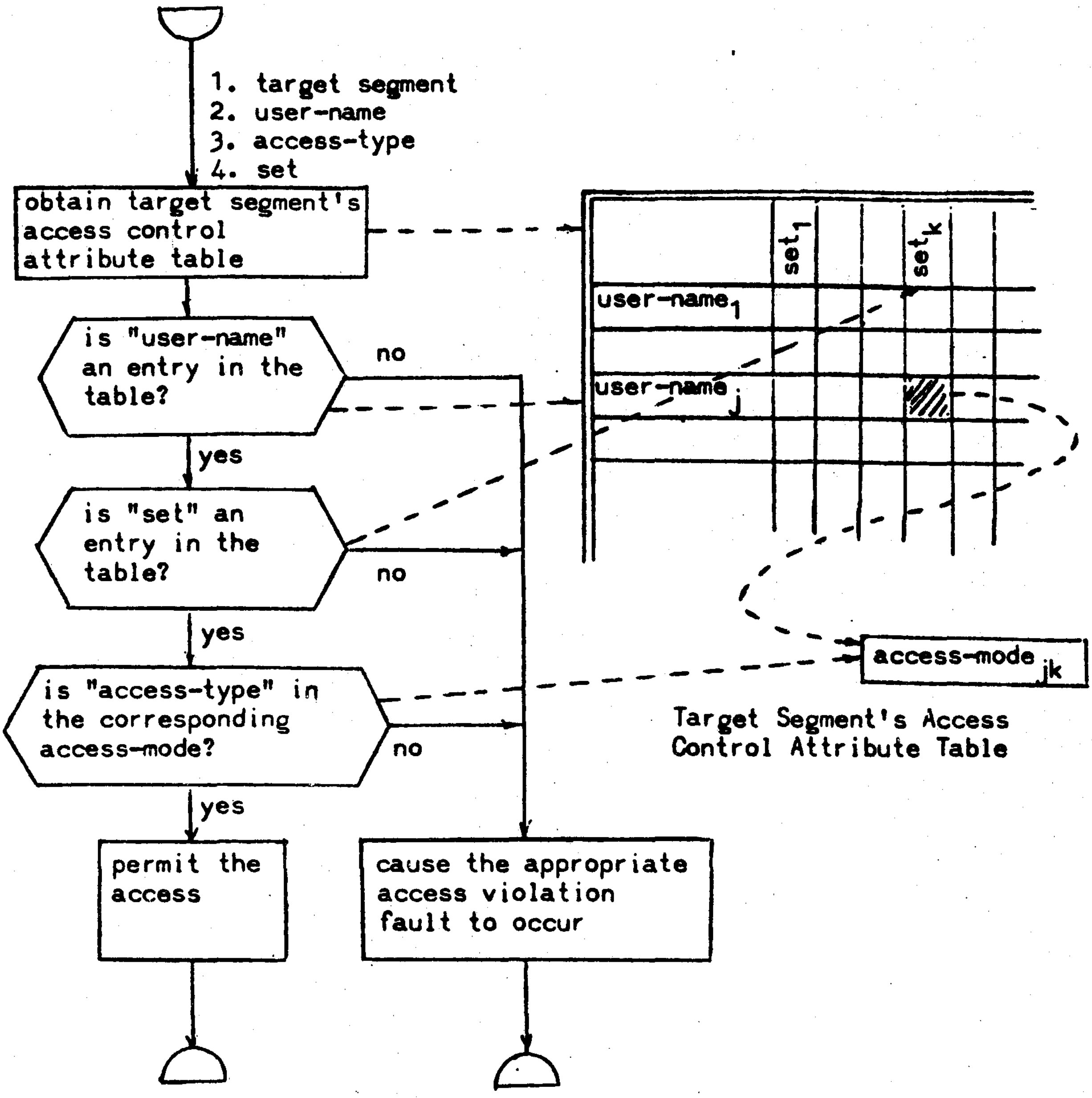


Figure 3. Illustrating an Access Control Mechanism Depending on User-name, Access-type, and Set

A corollary of this definition is that a user's access rights to a segment can in part be expressed as an access-mode and a triple of ring numbers - r(READ), r(WRITE), and r(EXECUTE) - indicating that a given access type, X, if present in the access-mode, is to be available to the user in the rings 0-r(X), inclusive. We shall defer to the next chapter consideration of how a process changes from one ring to another.

Figure 4 illustrates access control base on the ring in which the process is executing. The essential point to notice is that a segment's access control attributes can be very concisely recorded. The interpretation of the access control attributes is as discussed in the preceding paragraph.

A few comments about rings may be in order. First, the introduction of rings greatly simplifies the recording of a segment's access control attributes, as indicated above. Second, the fact that rings are ordered removes the ambiguity about the changing of power that was inherent in the idea of a transition between sets: when the processor changes from ring i to ring j, j>i implies an increase (or at least no decrease) of power or privilege with respect to all segments; and j<i implies a decrease. This homogeneous and evident change of power with the change of ring makes it much easier to think about the problems of changing rings than it could ever have been to think about the changing of sets. As we shall see, and notwithstanding the previous remark, most of the difficulty in the fully worked out strategy of access control by ring nevertheless resides in the mechanics of changing rings.

The following points seem to be necessary elements of any access control strategy based on the idea of rings:

- Attempts by the processor to pass control from one ring to another must be supervised by the access control mechanism.

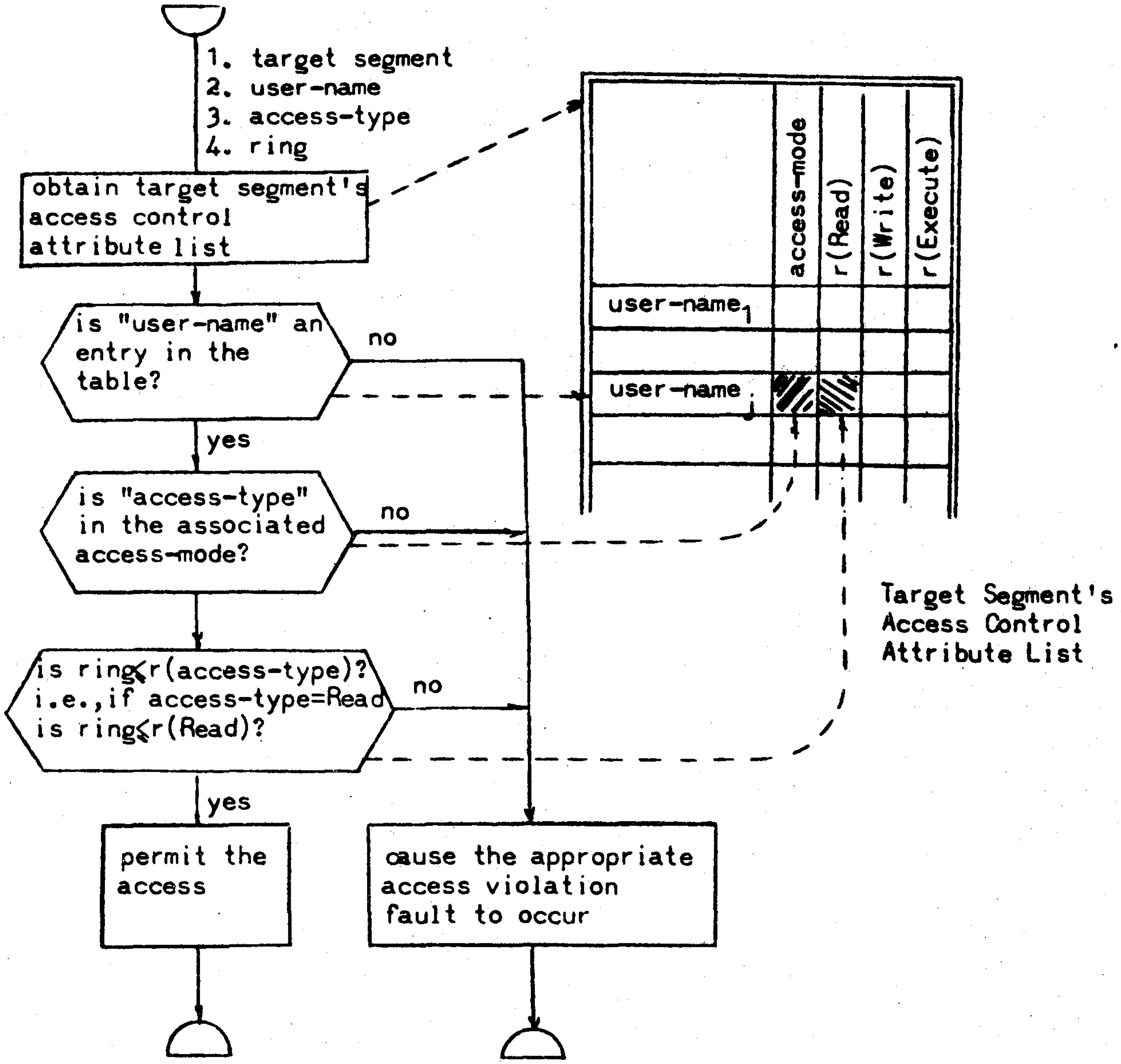


Figure 4. Illustrating an Access Control Mechanism Depending on User-name, Access-type, and Ring

- The transition from one ring to another can only occur upon a call or return; the transition (if any) associated with every call and return must be unambiguously defined.

- The concept of a "return" from a call must be extended to imply returning to the ring from which the call was made.

3.1 Multics Terminology; Gates

In Multics the term "inward" is used to characterize a transition from one ring to a more privileged ring, the term "outward" to characterize a transition in the other direction. Procedures which may be called by "inward" calls are called "gate" procedures since they are, in effect, gates through which the processor may enter the more privileged ring. We shall see that the major difficulties in the design of a ringed access control system relate to allowing outer ring procedures to use inner ring procedures without allowing them to defeat the protection purposes responsible for the existence of rings.

Chapter 2

MULTICS RING STRUCTURE PHILOSOPHY

In Chapter 1 we discussed access control and developed the idea of rings in a general (i.e., non-Multics) way. We now turn to Multics itself. In this chapter we shall enlarge on the subjects of rings and gates, state and justify the full Multics ring structure strategy, and show how this strategy can be implemented with the aid of suitable hardware.

In this and the following chapters, the emphasis in the definition of "ring" will shift slightly. We will think of a ring not as a process state defined by a set of procedures, but rather as an abstract process state in which, by virtue of the access control rules of the system, a particular set of procedures may be permitted to execute. There are 64 rings in Multics which are conventionally numbered, in order of decreasing power, from 0 to 63.

1. THE PRELIMINARY STRATEGY

A preliminary (and conceptually useful) idea for the use of rings is based on specifying a user's access rights to a given segment with an access-mode, a single ring number "r", and a gate-switch.

The Rules. The ring number r, gate-switch and access-mode are interpreted as follows. (Note that all ring intervals are inclusive).

a. If the user's access-mode contains WRITE, the user may, in rings (0,r), write in the segment.

b. If the user's access-mode contains READ, the user may, in rings (0,r), read the segment.

c. If the user's access-mode contains EXECUTE, the user may,

   1. in ring r, call and execute the segment

   2. in ring R<r, call the segment, switching to ring r to execute it

   3. in ring R<r, but only if the gate-switch is set, call the segment, switching to ring r to execute it

      Every attempt by the process to switch to a lower numbered ring in this way must pass a legitimacy test imposed by the access control mechanism and by the procedure being entered.

d. All ring switching must be done under the supervision of the access control mechanism.

e. The concept of "return from a call" must be extended to imply a return to caller's ring.

The Need for "Gates". Since an "inward" call (i.e., a call through a "gate") increases the processor's power, it is necessary that a test be made to verify that the process has attempted to enter the more powerful ring on a legitimate errand. For, if the process could freely change its ring so as to increase its power, the protection offered by the ring aspect of the access control mechanism would be wholly illusory. The kind of testing is occasioned by an attempted inward ring change will be discussed in detail in Chapter 3. As an obvious example, we note that a call to a gate segment should be permitted only if the target address is in fact an entry point of the segment.

2. THE "RING BRACKET" STRATEGY

The principal difficulty with the "preliminary" strategy described above is that procedure segments may be executed in one ring only. This means that a procedure likely to be called in several rings will often be called from a ring other than its ring of execution, occasioning a great deal of ring changing, an expensive business as we shall later see. A second difficulty with the "preliminary" strategy is that users with both READ- and WRITE-access rights for a segment have these rights equally in all of the rings from 0 to r. Since the ability to write in a segment is intrinsically more powerful than the ability to read it, it would be desirable to be able to grant write permission to a user in a (relatively privileged) subset of the rings in which he may read. As a result of these and other considerations, Multics has rejected the "preliminary" strategy for a "ring bracket" strategy.

Under the "ring bracket" strategy, a user's access rights respecting a given segment are encoded in an access-mode and a triple of ring numbers, (r1, r2, r3), called the user's "ring brackets" for the given segment.

<u>The Rules</u>. The ring brackets, (r1, r2, r3), which must satisfy the relations $r1 \leq r2 \leq r3$, are interpreted as follows. (Note that all ring intervals are inclusive).

a. If the user's access-mode contains WRITE the user may, in rings (0,r1), write in the segment.

b. If the user's access-mode contains READ the user may, in rings (0,r2), read the segment.

c. If the user's access-mode contains EXECUTE the user may,

   1. in rings (r1,r2) call the segment without changing ring

   2. in rings (0,r1-1), call the segment, switching to ring r1

   3. in rings (r2+1,r3), call the segment, switching to ring r2

      Every attempt by the process to switch to a <u>lower</u> numbered ring in this way must pass a legitimacy test imposed by the access control mechanism and by the procedure being entered.

d. All ring switching must be done under the supervision of the access control mechanism.

e. The concept of "return from a call" must be extended to imply a return to the caller's ring.

Under these rules we see that a utility routine may be given ring-brackets (0,63,63) and so be callable in all rings, but never occasion a change of rings upon being called. On the other hand, a critical system procedure might have ring brackets (0,0,0) and so be callable and executable only in ring 0.

We also see that a user who has read and write permission for a data segment may be given ring brackets (a,b,b) with $a<b$ so that the domain in which he has write permission, rings (0,a) is a relatively privileged subset of the domain in which he has read permission, rings (0,b). These comments show how the ring bracket strategy corrects the defects which we noticed in the preliminary strategy.

<u>Ring Changing Calls</u>. Let us now discuss inward and outward calls. The "rules" provide that every procedure segment for which $0<r1$ may be entered via an outward call (from ring 0, for instance) and that those procedure segments for which $r2<r3$ are "gate" segments and may, therefore, be entered via inward calls (from ring r3, for instance). What is the nature of such calls?

An inward call is made when a procedure in an outer ring wants to increase the power of its process temporarily in order to do a job requiring such increased power. For example, a user procedure may call a system procedure in ring 0. The notion of "inward call" brings to mind "the tail wagging the dog", since lesser power directs the use of greater power. The only segments which can be entered via inward calls are, therefore, the "gate" segments. The duty of a gate segment, as a gate segment, is to perform a test of the legitimacy of the inward call, that is, to see that the caller has not, by accident or design, asked the gate segment to behave irresponsibly. Whether or not a segment is a "gate" for a particular user depends on that user's ring brackets and access-mode respecting that segment.

An outward call is made when a procedure executing in an inner ring wants a job done which can (and perhaps must) be accomplished with the comparatively feebler power of an outer ring. For example, a process in Multics initializes itself (a system function) in ring 0 but calls out to a user ring when ready to do the user's work. In this case, the process must call out since a Multics convention forbids user work to be done in ring 0. For another example, a programmer with a collection of more or less debugged procedures may use several rings, keeping the more debugged procedures and their data in the inner rings so that damage from the other procedures will be isolated in the outer rings. If these procedures call each other freely, outward calls will presumably occur.

3. RECORDING AND RECOVERING ACCESS CONTROL RIGHTS

In "The Multics Virtual Memory"[1], we find that all of a segment's attributes of interest to the system are stored in the segment's "branch" in a "directory" segment. The access control attributes of a segment are stored in its branch in a variable length table called the access control list (ACL). Each entry of a segment's ACL specifies a particular user's access rights respecting the segment and is of the form:

user-name, access-mode, ring brackets

The procedure responsible for determining a user's access rights for a given segment searches that segment's ACL for the user's user-name. If it is not found, then the user has no rights. If it is found, then the user's access rights are determined by the associated access-mode and ring brackets.

4. "RINGED HARDWARE"

In "The Multics Virtual Memory"[1] we discussed the use of the 645's descriptor segment and Segment Descriptor Word (SDW) in providing the Virtual Memory. It was noted that part of each SDW was reserved for an access control field. In this section we shall discuss hardware similar to the 645's which is consistent with the description given in "The Multics Virtual Memory" and which permits a simply described implementation of the Multics ringed access control strategy. In Chapter 4, we shall describe the actual 645 hardware and discuss the software modifications needed to provide for the differences from the hardware described here.

We propose "ringed hardware" with the following features:

1. The processor has a ring register whose value defines the process' ring. This register may be changed by instructions only in ring 0, that is, when its value is 0.

2. The SDW's access control field contains the process' access-mode and ring brackets.

3. The processor has an access control mechanism which checks attempted memory accesses according to the rules stated in Section 2 and causes the processor to fault (trap) to an appropriate procedure in ring 0 in cases where the attempted access cannot be (or cannot be directly) performed. Such a fault causes the hardware to set the ring register to 0.

It should be clear that a procedure executing in ring n should not be able to change the value of the ring register to m, m<n, simply by executing an instruction. It might, however, seem that ring changing should be accomplished by the hardware itself, during the execution of a transfer instruction, by a simple change of the contents of the ring register. We have avoided specifying such hardware, for, as we shall see in Chapter 3, changing rings is quite complex and requires considerable software assistance. In light of this fact and considering the hardware organization described above, we may describe the functioning of this system as follows:

When a memory access is attempted, the type of access (read, write, or execute) and the processor's ring register are compared, by the processor's access checking mechanism, with the access-mode and ring brackets fields of the target segment's SDW. As a result of the comparison, three actions may be taken:

1. The memory access is performed and the ring register is unchanged.

2. The memory access is a ring changing transfer; the processor faults to the ring changing fault handler, in ring 0.

3. The access attempted is illegal; the processor faults to a suitable access violation fault handling procedure, in ring 0.

Note that the fault handling mechanism must have the power to change the ring register. This is achieved by making the fault handling procedure executable in ring-0 only, making the hardware enter ring 0 upon taking a fault, and making the ring register changeable (by instruction) in ring 0 only.

Chapter 3

SOFTWARE FUNCTIONS IN RING CHANGING

We indicated, in Chapter 2, that ring changing is a complex activity requiring considerable software assistance. In this chapter we will discuss various aspects of an operating system imposed by a ringed access control system and will discuss the software functions consequently required in ring changing.

We will structure our exposition by separately describing the four types of ring changes, inward and outward calls and returns, attending to points of interest as they arise. That done, we will conclude with a discussion of important facts and concepts and a quick once-over of the ring changing software.

Many of the functions to be described below might be performed, at least in part, in the inner-ring procedure involved in the change of ring rather than in the procedures of the ring changing mechanism, and some of the functions might more naturally be performed there. We take the point of view, however, that the code required to perform ring changes should be concentrated in a single place and we give the ring changing mechanism responsibility for performing all of these functions. In order to handle ring changing in this way, it is necessary to establish certain conventions between the ring changing mechanism and the inner-ring procedures involved in ring changes, as we shall see below.

1. INWARD CALLS

Detection. An inward ring changing call is detected when an inward ring changing fault occurs as the result of a "call" (rather than of a "return") type of instruction. The fault handler obtains the number of the target ring from the process' ring brackets for the target segment; according to the rules of Chapter 3, Section 2, the target ring is "r2".

Gate Address Validation. The handler's first business is to verify that the address to which the caller wishes to transfer is indeed a gate entry point for the process. This verification is based on a "gate-list" (i.e., a list of gate entry points) associated with the target segment; this gate-list may be system-wide in the sense that all users who may use the segment as a gate may use the same gate entry points or may be per-user in the sense that each user who may use the segment as a gate has a private gate-list. In today's Multics, the gate-list is system wide and is stored in the procedure segment (rather than, for instance, in the segment's access control attributes in the segment's branch).

Per Ring Data. We must now consider the nature of the "workspaces" of the calling and of the called procedures. In the ringed environment, all data must be "ring bracketed", including workspace data, e.g., the PL/1 static and automatic data. Since a procedure executing in ring r may freely copy into the (ring r) workspace any data readable in ring r, including all such data not readable in ring r+1, it is clear that ring r must use a workspace with ring brackets (r,r,r). Thus, assuming that any workspace segment has an access-mode implying read and write permission, the workspace for ring r is readable and writeable in rings 0 to r and cannot be accessed at all in the rings r+1 to 63. The above considerations imply that the process needs distinct workspace segments corresponding to the rings in which the process executes. Hence, the inward ring changing fault handler will have to provide the proper workspace for the called procedure.

The Environment. We may generalize from the idea of a workspace segment to the idea of an environment. Object procedures expect to execute and, therefore, to be transferred to, in a conventional environment defined by various appropriately valued hardware registers and data structures. Among other things, the environment specifies the workspace to be used by the procedure. (Thus, for example, Multics procedures expect certain processor base registers to be pointing to appropriate "stack" and "linkage" segments). Since this conventional environment is assumed, it is obviously a duty of the ring changing mechanism to create the environment which the procedure being entered will expect to use.

Argument Validation. Now let us consider the arguments which may be passed by the caller to the called (gate) procedure. To begin with, providing a suitably initialized environment for the called procedure involves copying the address of the argument-list (or copying the argument-list itself) into the environment of the called procedure. Certain precautionary measures then become necessary which relate to the need for a "gate" to act responsibly, as discussed in Sections 1 and 2 of Chapter 2. Let us motivate the discussion of these precautions by considering two examples of inward calls which should be aborted by a careful ring changing mechanism.

1. A ring-50 procedure calls a gate procedure in ring-32 and specifies a return argument in the workspace of ring-40. If the call is not aborted, the gate procedure may write in the ring-40 segment at the explicit request of the ring-50 procedure. The gate procedure would thus in effect permit the ring-50 procedure to overwrite the ring-40 segment, a clear violation of the access control philosophy.

2. A ring-50 procedure calls a gate procedure in ring-32, specifying return arguments in ring-50 segments and input arguments in ring-40 segments. If the call is not aborted, the gate procedure may copy ring-40 data into ring-50 segments. The gate procedure would thus in effect permit the ring-50 procedure to read ring-40 data, another violation of access control philosophy.

The responsibility of a gate procedure may be characterized as avoiding the improper use, on behalf of an outer ring procedure, of that part of its accessing power which exceeds that of its caller. To fulfill this responsibility, a gate procedure must, before accessing memory via an address obtained from its caller (or from any other outer ring source), verify that the intended type of access could have been performed by the caller. We shall refer to this as "validating the address". Once all addresses obtained from outer rings have been validated, the gate procedure may freely proceed, since it is clearly safe to use all other addresses.

Although this "address validation" can all be done by the gate procedure itself, our point of view suggests that as much of it as reasonably possible be done by the ring changing mechanism. Since most of the addresses supplied to a gate procedure by its caller are the addresses of arguments, we assign the business of validating these addresses to the ring changing mechanism and we leave it to the gate procedure itself to validate all other suspect addresses. Checking the addresses of the arguments is called "argument validation". Argument validation should include checking that the caller has READ-access for all of the arguments being passed and WRITE-access for those arguments, including "return" arguments, in which the called procedure may write. Argument validation implies a further step in inward ring changing: argument-list copying. For, if a pointer is checked to see that its value may safely be used, then the pointer may not safely be left in a segment where it may be changed by a process executing in a ring less privileged than the gate's. Therefore, all addresses to be checked must be copied into the gate segment's workspace prior to such checking.

It is clear that the argument validation mechanism must make use of an argument-list-descriptor, presumably coded as data, associated with the gate entry point. This descriptor tells how many arguments are expected and how they are to be used (i.e., whether they will be read and/or written in).

The argument-list-descriptor(s) for a gate segment may be implemented in many ways, for example, as part of the gate segment's gate-list. In any case, it is clear that the argument-list-descriptor, like the gate-list, must be supplied to the ring changing mechanism by the gate (inner-ring) procedure rather than by the calling procedure.

Figure 5 illustrates a gate procedure segment, with its gate-list and argument-list-descriptors, in a straightforward implementation. When this segment is called from an outer ring, the ring changing mechanism validates the attempted transfer address by finding it on the gate-list and validates arguments by checking that the caller has the access rights toward the arguments which are specified in the argument-list-descriptor associated with the transfer address.

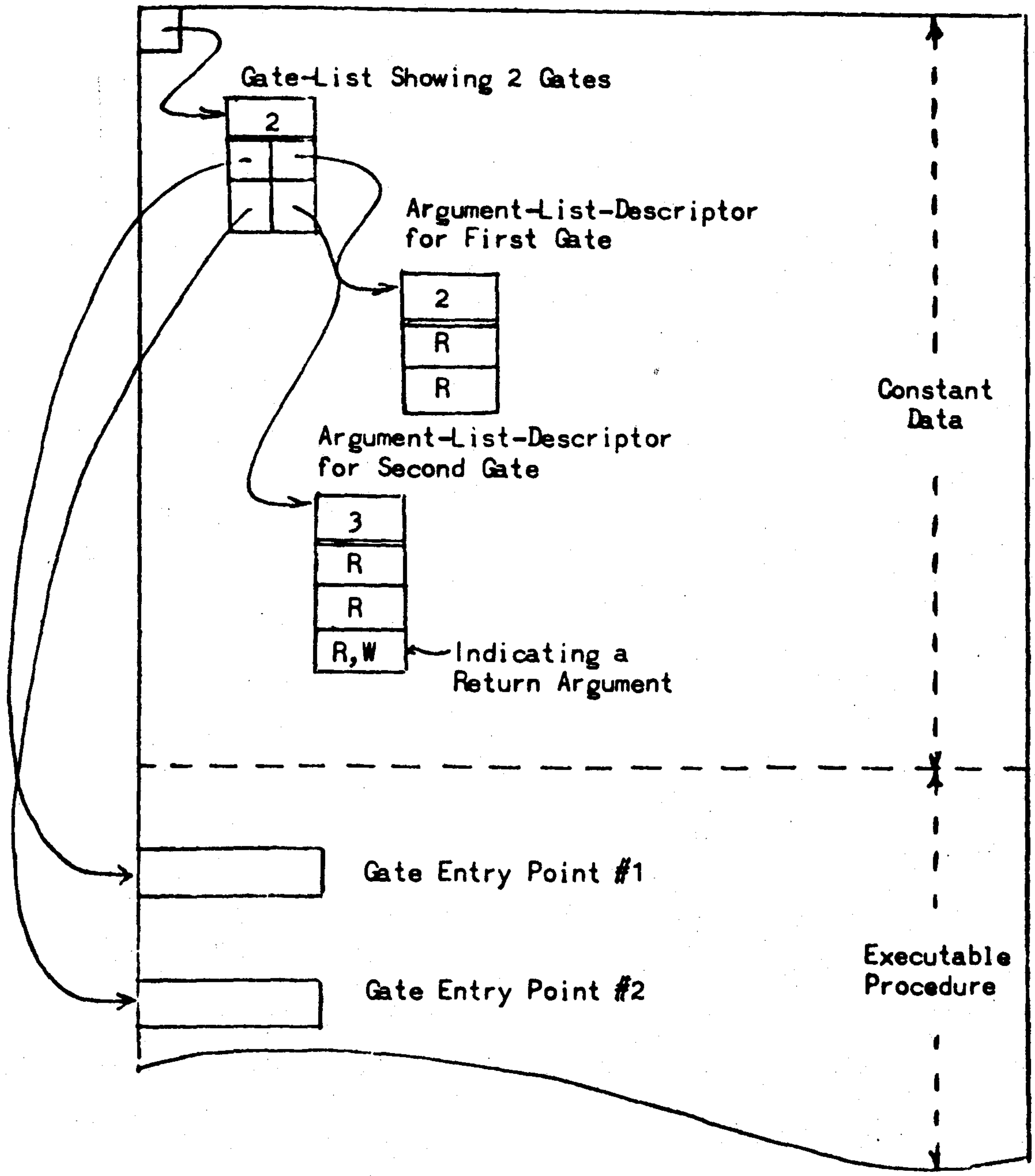


Figure 5. A Gate Segment Showing Gate Entry Points, Gate-List and Argument-List Descriptors

2. OUTWARD RETURNS

The Detection Problem. The detection of an outward return is not straightforward. Since the procedure to be returned to may well have ring brackets permitting it to execute in the returner's ring and, indeed, in a number of other rings, one may well wonder how the ring changing fault associated with the return is generated and how the ring changing mechanism decides which ring to return to.

An example may make these remarks clearer. Consider a call from ring-20 to a procedure P with ring brackets 5-10-20 and a call by it to a procedure Q with ring brackets 3-7-12. The first call takes the process into ring-10 and the second takes it into ring-7. It is clear that an ordinary return from procedure Q would not cause a ring changing fault. It is also clear that if it did cause a fault, the fault handler would have to choose a ring to return the process to from the interval ring-5 to ring-10.

Forcing A Fault. We see that in the case of a ring changing return, the ring bracket mechanism cannot by itself be dependent upon to cause the necessary ring changing fault or to provide the information required to identify the caller's ring. A special trick is, therefore, used to cause the fault. The normal return pointer in the returner's workspace is over-written with a conventional replacement so that when the process attempts to return via this "return pointer", a fault will occur which is associated with the ring changing return fault handler. This device for forcing a return fault applies equally to inward returns and is also used in that case.

The Return Stack. When the artificial ring changing return fault occurs, as a result of a "return" type of instruction, the ring changing return mechanism is invoked. It must not look in the returner's workspace to find the information that it needs to perform the return - caller's ring number and the return pointer - for these items could be manufactured by the "returner" to imply a "return" to a more privileged ring. The ring changing mechanism, therefore, maintains a ring-0 data base called the "return stack" in which it records all the information needed to perform all uncompleted ring changing returns, both inward and outward. At any time, the last entry on this stack specifies the return from the ring in which the process is then executing. We may now say that an outward ring changing return is a return which causes a ring changing return fault and whose entry in the return stack indicates a return to an outer ring.

Address and Argument Validation. There is no need, from an access control viewpoint, to validate a return address for an outward return since an inner ring procedure may in any case freely enter an outer ring at any point. However, to protect against error, the return pointer recorded in the return stack may be compared against a "validation return pointer" stored in the returner's workspace. Both the validation return pointer and the return pointer in the return stack would be recorded at the time of the corresponding call by the ring changing mechanism. If these return pointers disagree, then the ring changing return can be regarded as an error and treated accordingly.

There is no need for argument validation at the time of an outward return; the work was done at the time of the corresponding call.

The Restoration of the Environment. Finally, let us note that it is necessary, in servicing an outward return, to re-establish the environment that existed at the time of the corresponding call. The caller's workspace must be re-established, base registers must be restored, the entry on the return stack must be removed, etc. Any information which may be needed for this work must be found in the return stack entry for this return and must thus have been stored there at the time of the call.

3. OUTWARD CALLS

An outward ring changing call is detected when an outward ring changing fault occurs as the result of a "call" type of instruction. The ring to be entered is determined from the target procedure's ring brackets; according to the rules of Chapter 2, Section 2, the target ring is "r1". There is no need to validate the target address of the call, for gates govern inward calls only. As with the inward call, there is a need to establish the environment required by the called procedure.

Argument Copying. If, as is usual, the caller's arguments are stored in the caller's workspace, the arguments will be inaccessible to the called procedure in its outer ring. It is, therefore, insufficient to copy only a pointer to the argument-list or the argument-list itself into the workspace of the called procedure. It is necessary to copy the arguments themselves. This in turn implies that a new argument-list must be fabricated in the workspace of the called procedure which contains the addresses of the local copies of the arguments.

There is no need to perform access validation on the arguments. The inner ring procedure may judge for itself what data to pass to the outer ring. The copying of arguments is done, of course, with the authority of the calling procedure's ring, if not by the calling procedure itself. If the copying is actually done in a ring more privileged than the caller's, e.g., by the ring changing fault handler (which executes in ring-0), then the arguments must be access validated to make sure that no data are copied into the workspace of the called procedure to which the caller itself does not have access.

Note that argument copying depends on information, represented as a "copying descriptor", associated with the outward call (see Figure 6). The copying descriptor tells how many arguments there are, how they are to be used (i.e., whether or not they are to be written into), and what their lengths are (so that they can be correctly copied). We will discuss the question of arguments which are to be written into by the called procedure in the following section.

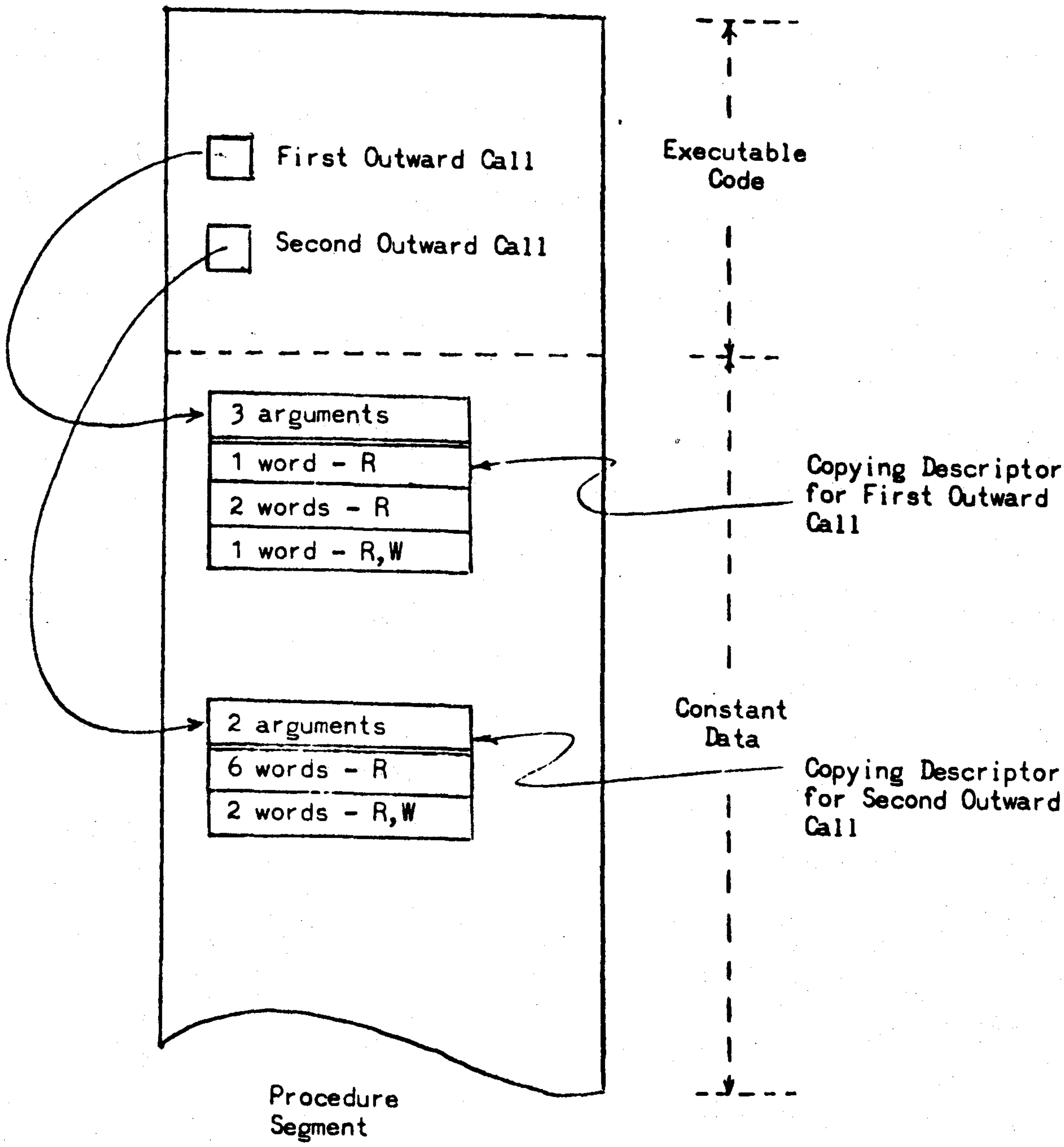


Figure 6. Example of a Procedure which makes two Outward Calls Showing Copying Descriptors

4. INWARD RETURNS

Inward returns are detected when the artificial ring changing fault occurs (see the discussion of outward returns in Section 2) and the return stack entry indicates an inward return. The return pointer in the return stack entry may be compared with a validation return pointer in the returner's workspace in order to avoid erroneous ring changing returns.

The arguments which the outer ring procedure may have written in, as identified in the copying descriptor, are then copied from the returner's workspace into the locations specified for them in the caller's (original) argument-list. Validation of these addresses is only necessary if the copying is done in a ring more privileged than that being returned to, e.g., by the ring changing fault handler which executes in ring 0.

Once these arguments have been copied, the ring changing mechanism re-establishes the environment of the calling procedure and returns to it.

5. REVIEW AND DISCUSSION

Detection. A ring changing transfer is detected when the ring changing mechanism is invoked in response to a suitable fault. The ring bracket mechanism (i.e., a mechanism respecting the rules set forth in Chapter 2, Section 2) will produce such a fault in the case of inward and outward calls; such calls are in fact so defined. Ring changing returns, though, are defined as returns from ring changing calls and the ring bracket mechanism cannot be depended on to detect these returns. The strategy of the "artificial ring changing return fault" was introduced (see Section 2) to guarantee that these returns would always be detected.

Transfer Address Validation. The basic fact about a ringed access control system is that a process' power depends on the ring in which it executes. This is meaningful only because of the rules which govern inter-ring transfers. The basic rule is that outward (power decreasing) transfers may be made at the procedure's discretion whereas inward (power increasing) transfers may be made only with "the permission of the ring to be entered". Transfer address validation, which consists of making sure that the target address is an address at which the target ring will permit entry, thus applies only to inward transfers.

In the case of an inward call, the target address is validated by finding it on the target procedure's gate-list, that is, finding it to be the address of a gate entry point. In the case of an inward return, the target address (which is obtained from the return stack) is validated implicitly by virtue of the fact that it was earlier supplied to the ring changing mechanism by the outward calling procedure, the very procedure being returned to.

The Return Stack. The return stack was introduced (see Section 2) as the data base in which the ring changing mechanism stores the ring number and return address of a caller so that the ring changing return mechanism can subsequently validate the return. The return stack must thus be accessed by the ring changing mechanism upon every ring changing call and return, being "pushed" at each such call and "popped" at each such return. To the extent that the ring changing mechanism may profit from storing other information from the time of a call to the time of the corresponding return, the return stack is evidently the "right" data base to use. Without going into detail we suggested, for example, that the return stack was a good place to record information needed for the restoration of the calling procedure's environment.

Argument Validation. Whenever an inner ring procedure accesses memory via an address obtained from an outer ring source, there is the danger that the supplier of the address is "using" the more privileged procedure to "get around" access control restrictions. Addresses obtained from outer rings are, therefore, suspect and must be used with discretion.

The most outstanding examples of suspect addresses are the addresses of arguments associated with inward calls. "Argument validation" is a technique by which the ring changing mechanism, acting on behalf of the class of gate entry points, does a standard and generally sufficient job of checking these addresses.

Argument validation is not only used in the case of inward calls but in the case of those outward calls where arguments are copied as well. When the arguments for an outward call are copied into the workspace of the called procedure and later, when the return arguments are copied back into the workspace of the calling procedure, the copier of these arguments, being part of the ring-0 ring changing mechanism, obtains its arguments from the rings of the calling and called procedures and must validate these addresses.

Although argument validation doesn't handle all cases of "suspect" addresses, the existence of argument validation does have the useful effect of isolating the cases which aren't covered, making life easier for the programmer of a gate procedure. For, if he can make sure that all of the suspect addresses to be used by the gate procedure and its dynamic descendents are the addresses of arguments, he may be assured that he has written a proper gate procedure. And if there are a few other addresses requiring checking, he can handle them on a case by case basis.

6. OUTLINE OF RING CHANGING SOFTWARE

A. Inward Calls

1. Check that the specified address is a gate entry point.

2. Store information in the "return stack" specifying the caller's environment, including caller's ring number and the return address specified by caller.

3. Determine the ring (NEW-RING) to be entered; that is the value r2 from the called procedure's ring brackets.

4. Create an environment for the called procedure in NEW-RING.

5. Copy the addresses of the arguments into the environment of the called procedure and perform "argument validation".

6. Associate a ring-changing-return fault with the normal return from the called procedure.

7. Set the ring register to NEW-RING.

8. Perform the call.

B. Outward Returns

1. Check that this return corresponds to the last entry in the "return stack".

2. Clean up the environment of the returning procedure (undo A-4).

3. Determine the ring to be returned to, OLD-RING, from the "return stack".

4. Restore the caller's environment, as specified in the "return stack".

5. Set the ring register to OLD-RING.

6. Return to the caller at the address specified in the return stack.

C. Outward Calls

1. Store information in the "return stack" describing the caller's environment.

2. Determine the ring, NEW-RING, to be entered; this is the value r1 from the called procedure's ring brackets.

3. Create an environment for the called procedure in NEW-RING.

4. Copy the caller's arguments into the new environment and create an argument-list pointing to the copied values, also in the new environment.

5. Associate a ring-changing-return fault with the normal return from the called procedure.

6. Set the ring register to the value NEW-RING.

7. Perform the call.

D. Inward Returns

1. Check that this return corresponds to the last entry in the "return stack".

2. Determine the ring, OLD-RING, to be returned to from the "return stack".

3. Copy the return arguments back into the caller's environment (in OLD-RING).

4. Clean up the returning procedure's environment.

5. Restore the caller's environment, as specified in the "return stack".

6. Set the ring register to OLD-RING.

7. Return to caller at the address specified in the return stack.

Chapter 4

SIMULATION OF RINGS USING THE 645

The 645 differs from the "ringed hardware" described in Chapter 2 in several respects which, taken together, add up to the fact that the 645 is a 2-ring rather than a 64-ring machine. In this chapter we shall discuss the relevant aspects of the 645 hardware and show how the ringed access control strategy described in Chapters 2 and 3 can be simulated on the 645.

1. FEATURES OF THE 645 NEEDED FOR THE SIMULATION

1.1 The 645 does not have a "ring register" but does have two states, called master mode and slave mode. The processor has greater power when in master mode than when in slave mode; in particular, (a) certain instructions can only be executed when the processor is in master mode and (b) the access control field of the 645's SDW permits the specification, in addition to the access-mode, of a limiting descriptor - "accessible in master mode only."

1.2 The access control field of the 645's SDW contains no information about rings; in particular it does not contain ring brackets. It does, however, contain either:

a. access-mode information possibly including either of the two descriptors:

   - accessible in master mode only
   - master mode procedure

b. the specification of one of eight special "directed" faults (traps) which is to occur whenever the SDW is accessed.

The processor is only "in master mode" when executing a procedure whose SDW indicates a "master mode procedure." The processor may enter master mode while executing a slave mode procedure by:

- faulting
- taking an interrupt

There is another way of switching from slave mode to master mode; it will be discussed later since it invokes a hardware feature that is not needed to simulate a ringed machine.

1.3 The 645 processor's access control machinery interprets the SDW during the addressing cycle and causes an appropriate action to occur depending on the SDW and (usually) on the attempted access, as follows:

a. If the SDW implies a particular "directed fault", then that fault occurs.

b. Otherwise, if the SDW does not permit the attempted access, the appropriate access violation fault occurs.

c. Otherwise, the SDW permits the attempted access and the access is performed.

When a fault occurs, the 645 enters master mode and transfers control to the appropriate master mode fault handling procedure.

1.4 Among the instructions which are "master mode only" are those which access the processor's DBR (the Descriptor Base Register, which contains the absolute address of the descriptor segment currently in effect) and all I/O connect instructions.

2. SIMULATING THE "RINGED HARDWARE" ON THE 645

The technique of simulating the "ringed hardware" on the 645 can practically be deduced from the requirements of that simulation:

1. It must be possible to simulate being in a given ring.

2. It must be possible to simulate changing from one ring to another.

To simulate being in a ring, it must be possible to set up a 645 descriptor segment to define the same set of potential actions in response to potential attempted accesses as is defined by any given "ringed descriptor segment, ring register" pair. The potential actions will be the same in the 645 as on the "ringed hardware" if (a) permitted accesses are performed by the 645 without causing a fault and (b) if accesses which would cause a fault on the "ringed hardware" cause a fault on the 645.

To simulate changing from one ring to another it is obviously necessary to be able to change the 645 descriptor segment. This may be done in two ways. If space is felt to be at a premium, the 645's master mode fault handlers may "change rings" by over-writing the existing descriptor segment with the values appropriate to the other ring. On the other hand, if processing time is felt to be more important than space, the fault handler in master mode may "change rings" by altering the DBR to point to that descriptor segment (waiting in the wings, so to speak) which corresponds to the ring being entered. This second technique, used in Multics, requires one descriptor segment per-ring for each process. The per-ring descriptor segment thus becomes part of the "environment" which pertains to each ring, and switching descriptor segments becomes part of the job of the ring changing mechanism.

3. AN ADDITIONAL FEATURE OF THE 645

The 645 processor has the ability of switching from slave mode to master mode without invoking the trap mechanism, as follows:

A slave mode procedure can transfer to a master mode procedure M provided that:

a) the segment descriptor of M contains the "accessible in slave mode" attribute, and

b) the transfer be directed to location zero of M.

This technique for increasing a process' power differs from ring changing in the sense that no fault is generated. However, the philosophy remains the same. By checking that conditions (a) and (b) are true, the hardware performs the "gate validation". The fact that the transfer is guaranteed to be into location zero permits one to code explicitly any type of subsequent validation in the procedure M and to guarantee that the validation code will be executed. (The only system responsibility is to make sure that the transfer is directed to a gate; the gate procedure must take care of the rest.) This feature is used in Multics as explained below.

4. MASTER MODE AND SLAVE MODE IN RING ZERO

Master mode is the most powerful state of the 645 processor; ring zero is the most powerful state of the ringed processor simulated on the 645. It should follow that executing in ring zero means executing in master mode, and it would so follow, were it not for the 645 feature discussed in Section 3 above. That feature is used in order to permit the ring zero supervisor to execute partly in master mode and partly in slave mode, easily switching from one mode to the other.

The Multics ring zero can be regarded as being itself composed of two concentric rings. The more powerful or "master ring 0" contains all master procedures and also all data accessible only in master mode. The less powerful or "slave ring 0" contains all slave procedures and also all data accessible in slave mode. Going from slave ring 0 to master ring 0 can be done through gates provided by master ring 0; these gates are in fact master procedures accessible in slave mode, with the entry point at location zero of the segment. This technique permits efficient switching between slave and master mode in the supervisor and this is the motivation for this additional hardware feature in the GE 645.

Two questions are raised by this discussion. First, why don't "ring zero" and "master mode" coincide? And second, why isn't the special mechanism for entering master mode more generally used?

The supervisor should use master mode only for jobs requiring its special power. To use it for other purposes would increase the chance of disastrous errors due to hardware and software bugs since, for example, all I/O connect instructions are executable in master mode only. But since the supervisor must use master mode fairly frequently, it is desirable that the supervisor have a way of entering master mode which involves just enough validation to prevent accidental entry. Thus the special mechanism. And thus the restriction of the use of the special mechanism to "slave ring zero."

BIBLIOGRAPHY

1. Bensoussan, A. et al. The Multics Virtual Memory T.I.S. R69LSD3.

2. Evans, D.C. and Leclerc, J.Y. Address Mapping and the Control of Access in an Interactive Computer SJCC 1967

3. Graham, R.M. Protection in an Information Processing Utility, CACM (May 1968)

4. Organick, E.I. A Guide to Multics for Subsystem Writers Chapter IV, Access Control and Protection Multics Repository Document M0106.

C. Series 6000 Features for the Multics Virtual Memory

CONTENTS

ILLUSTRATIONS

INTRODUCTION

Experience to date with the Model 645 hardware and the Multics software has uncovered many areas in which system performance and maintainability could be substantially improved by certain modifications of the 645 specifications. Four areas have been investigated and shown to be of major importance to the performance of Multics. The Multics extensions of Series 6000 processors both allow execution of Multics on these processors and provide considerable improvements in system performance. These features include:

1. Hardware aids for improved compatibility with other product line software.

2. Refinements of the paging and segmentation hardware to improve the performance of the system software.

3. Implementation of Multics ring protection mechanism entirely in hardware to improve system performance and reduce software complexity.

4. Addition of instructions for string manipulation and decimal arithmetic.

HARDWARE COMPATIBILITY

The issue of compatibility with product line software is really two issues stemming from two distinctly different motivations. One issue is that of "stand-alone compatibility," which allows the running of standard product line software (e.g., GCOS, T and D monitor) on a stand-alone machine. The other issue is that of "slave program compatibility," which allows for the efficient execution of Series 600/6000 slave programs under the control of the Multics system. A compatibility switch on the processor is used to handle the problem of stand-alone compatibility, while a program-settable mode is used as a software aid in handling the problem of slave program compatibility. The issues of stand-alone and slave program compatibility are treated separately in the following discussion.

MODIFICATIONS OF PAGING AND SEGMENTATION HARDWARE

This paper describes a number of modifications of the 645 paging and segmentation hardware which improve the performance of the Multics software and simplify some areas now felt to have been overdesigned in the 645. The changes in 645 specifications are summarized below (and are described in detail later in the paper):

1. The address field in the segment descriptor word is extended to a full 24-bit absolute address to allow page tables (and unpaged segments) to begin on any legal memory address. This modification allows the software a great deal more flexibility in the allocation of page table space and greatly reduces the amount of wired-down core storage reserved for page tables.

2. The processor supports only a single page size rather than the two page sizes supported by the 645. Provision is made to allow the page size to be modified by field engineering in an orderly and well understood fashion.

3. Each of the eight pointer registers of the processor is extended to contain both a segment number and a word number portion. The 645 concept of internal and external base registers and control fields is dropped. Each of the eight pointer registers on the processor behaves as a 645 base register pair. In addition, each of the new pointer registers contains a bit offset field for use by new instructions for string manipulation and decimal arithmetic.

4. Some minor changes have been made in the definition of the 645 master and slave modes, which are renamed respectively as the privileged and unprivileged modes to avoid confusion with existing terminology. A minor change has also been made in the treatment of execute-only segments, to allow entry at locations other than zero.

5. The access control information contained in the 645 page table word (not used in Multics) has been removed. In the new processor, all access control is implemented in the segment descriptor word.

HARDWARE IMPLEMENTATION OF MULTICS RING PROTECTION

The Multics concept of protection rings, ring crossing, and argument validation has been implemented as an integral part of the paging and segmentation hardware on the processor. The hardware implementation of rings is really a further modification of the 645 paging and segmentation hardware. However, the modification is introduced separately at this point since it involves perhaps the most significant deviation from the 645 specification and, as such, deserves somewhat more motivation. In the current version of Multics running on the 645, the ring protection mechanism is, of necessity, completely simulated by the Multics software. The current system maintains, in parallel, separate descriptor segments for each ring of each process. The ring crossing is simulated by a rather costly and complex fault processing mechanism which includes the copying and validation of argument pointers and the switching of descriptor segments to simulate the effect of switching rings. The cost of the current simulation amounts to approximately 10-20 percent of the useful chargeable CPU time and contributes substantially to the overall complexity of the system. In the new processor, ring crossing and argument validation are handled directly by the hardware without costly software intervention. As a result, a call to an inner ring will require no more CPU time that a call to a procedure in the same ring.

INSTRUCTIONS FOR STRING MANIPULATION AND DECIMAL ARITHMETIC

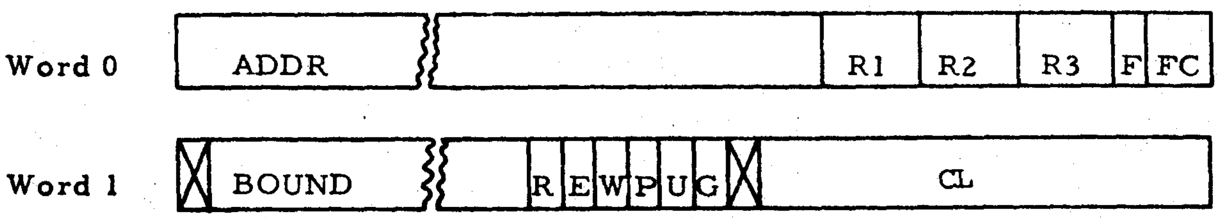
Extension of the 645 instruction set to include instructions for string manipulation and decimal arithmetic allows considerable simplification of both supervisor and user programs. The Series 6000 Extended Instruction Set (EIS) provides commands to directly process bytes, BCD characters, packed decimal data, and strings. The supervisor will take full advantage of the savings allowed by these new instructions. The language compilers will also take advantage of these space and time saving instructions.

SEGMENTATION AND PAGING IN NEW PROCESSOR

This section describes in detail the segmentation and paging hardware for the new processor. In most respects, the mechanism is quite similar to the 645 appending hardware, with the addition of some refinements to improve the performance of the system software. The single substantial deviation from the 645 specification is the addition of hardware to implement the Multics ring protection mechanism.

Segment Descriptor Word

In order to accommodate the hardware-implemented ring crossing and argument validation and other changes, the Segment Descriptor Word (SDW) has been extended to a 72-bit double precision word to be interpreted as described below.



ADDR (0-23) — Is a full 24-bit absolute address and specifies the core address of either a page table (for a paged segment) or the first location of an unpaged segment.

R1 (24-26) — Specifies the highest ring number of the read/write bracket for this segment (0-R1)[1].

R2 (27-29) — Specifies the highest ring number of the read/execute bracket of this segment (R1-R2)[1].

R3 (30-32) — Specifies the highest ring number of the call bracket of this segment ((R2 + 1) - R3)[1].

---

[1] See following section on Rings and Ring Brackets.

F (33) — Is a directed fault indicator and if off (=0) specifies that the processor is to execute the directed fault specified in the FC field (see below).

FC (34-35) — Indicates (if F is off) which of the four directed faults (DF0-DF3) the processor is to execute.

BOUND (1-14) — Is the boundary field and indicates the highest 16-word block of the segment which can be addressed without causing an out-of-bounds fault. If the high order 14 bits of an address to this segment is greater than the value of the boundary field, an out-of-bounds fault is generated. A boundary field of 14 bits is chosen because some instructions (e.g., the new version of STB) reference up to 16 contiguous words. (The boundary field could be maintained to the nearest word, but special checks would have to be made for instructions which reference two or more contiguous words.) A further implication is that the software is expected to allocate unpaged segments in a zero mode 16-word boundary.

R (15) — Is the read-permit indicator. Data fetches by other segments are permitted to this segment only if this indicator is on (=1) and if the processor is executing in a ring less than or equal to R2 (i.e., within the read/write or read/execute bracket).[1]

E (16) — Is the execute-permit indicator. Instruction fetches from this segment are permitted only if this indicator is on (=1) and if the processor is executing in a ring greater than or equal to R1 and less than or equal to R2 (i.e., within the read/execute bracket; see below). Note that when the E indicator is on and the R indicator is off, the segment is to be treated as an "execute-only" procedure segment. An execute-only procedure segment is permitted to reference data within itself (i.e., within the same segment) in spite of the lack of the read indicator. However, read permission is denied to other segments.

W (17) — Is the write-permit indicator. Attempts to store into this segment are honored only if this indicator is on (=1) and if the processor is executing in a ring less than or equal to R1 (i.e., within the read/write bracket; see below).

---

[1]See following section on Rings and Ring Brackets.

P (18) Is the privileged mode indicator. If this indicator is on (=1) and if the processor is executing in ring 0, the procedure segment is permitted to execute privileged instructions and inhibit interrupts under control of bit 28. Privileged procedures need no further powers and are subject to all other access checking (read, write permission bounds checking, etc.). Since privileged procedures can be executed only in ring 0, it is no longer necessary to limit calls to privileged procedures to enter via word 0 of the segment.

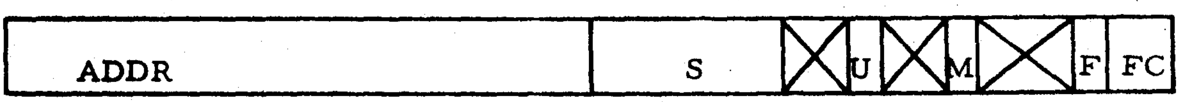
U (19) Indicates whether the segment is paged (=0) or unpaged (=1). If the segment is unpaged, ADDR is the full absolute address of the first word (word 0) of the segment. If the segment is paged, ADDR is the full absolute address of the beginning of the page table for the segment.

G (20) Is the gate indicator and if off (=0) any call to this segment from a different segment must be directed to an address value less than the value of the CL field (see below).

CL (22-35) Is the call limiter. If G is on, any external transfer to this segment via the new CALL instruction (described below) must be directed to a word number less than the value of this field.

Page Table Word

The format of the page table word (PTW) has been somewhat simplified from the 645 version in that no access control is performed at the PTW level. The PTW format is described below.



ADDR (0-17) Is the high order 18 bits of the 24-bit absolute address of the first word of the page. The hardware assumes that all pages begin on addresses which are zero modulo the page size. For example, if the page size is set to 1024 words, the hardware assumes that each page begins on a zero modulo 1024 address and that the low order 10 bits of the 24-bit absolute address are zero.

S (18-23) Is reserved for the use of the system software for the maintenance of page status information and is never modified (or used) by the hardware.

U (26) Indicates whether (=1) or not (=0) the page has been used since the last time this bit was interrogated (and reset) by the system software. Whenever this bit is zero and the processor addresses any word within the page (corresponding to this PTW), the processor sets this bit to 1 using a 3-bit store-by-zone command for bits 24-26. The store-by-zone is used to avoid a race condition with another processor attempting to set the "modified" bit (see below) for the same page. This technique is necessitated by the lack of a read-alter-rewrite command in the memory controller. A further implication of the lack of read-alter-rewrite is that the software must reset this bit via a store to the third 9-bit field (character) in the PTW in order not to disturb the modified bit. Note that any usage of the page between the time the used bit is read by the software and then reset (if on) is not noticed by the software. Since the used bit is used only for maintaining the core-usage statistics, this race between hardware and software has no effect, insomuch as (1) if the page is heavily used (i.e., needed in core) it will be used again turning the used bit back on, and (2) the software does not reset the bit if it is already zero.

M (29) Indicates whether (=1) or not (=0) the page has been modified since the last time this bit was interrogated (and reset) by the system software. Whenever this bit is zero and the processor modifies any word within the page, the processor sets this bit and the usage bit to 1 using a 6-bit store-by-zone command for bits 24-29. The software uses this bit to determine whether or not the contents of a page must be written on secondary storage before the core is released for other usage. As a result, the software is expected to store a directed fault in the PTW and clear the associative memories of all processors before the modified bit may be safely tested and then reset. In addition, the directed fault must be stored using a store to the sixth 6-bit field (character) of the PTW to compensate for the lack of read-alter-rewrite.

F (33) Is the directed fault indicator and if off (=0) indicates that the directed fault indicated in the FC field is to be executed by the processor.

FC (34-35) Indicates (if F is off) which of the four directed faults (DFO-DF3) is to be executed by the processor.

Modifying the Page Size

As indicated by the format of the SDW, the new processor supports only a single page size (the 645 allows two page sizes). However, it is extremely desirable to have the ability to change the page size in order to allow system optimization with respect to core "breakage" and storage device access times. For example, replacing the current highspeed drum with a bulk core would most likely give even better performance with a page size smaller than 1024 words.

Since a decision to change the page size is not a casual one and should not be made very often, the page size is changeable by field modification to any power of 2 from 64 words to 4096 words.

Absolute Address Formation

For each memory reference we assume the program presents the processor with the following address:

SEGNO	WORDNO

SEGNO (15 bits) Specifies the desired segment (i.e., index into the descriptor segment). The segment number is constrained to 15 bits (rather than 18) by considerations described later in this document.

WORDNO (18 bits) Specifies the desired word address (18 bits) within the specified segment.

We also assume (for discussion only) that the processor has the following two internal registers:

PN

PO

PN (12 bits) Is used to hold the page number (i.e., index into page table) when forming an absolute address within a paged segment.

PO (12 bits) Is used to hold the offset within the page when forming an absolute address within a paged segment.

PN is initialized with the high-order portion of WORDNO to obtain an index relative to the base of the page table of the segment. PO is initialized with the remaining portion of WORDNO and is augmented by the ADDR field of the PTW to form the absolute address. The "break" in WORDNO is determined by the current page size. For example, if the page size is 1024 words (initial setting), the high order 8 bits of WORDNO are used to initialize PN and the low order 10 bits of WORDNO are used to initialize PO.

Figure 1 summarizes the major steps necessary to transform a program generated address (SEGNO/WORDNO) into an absolute address (ABSADDR). In most respects, the address formation is simpler than the 645 mechanism, in that there is only one page size to consider and that no access control is specified at the PTW level.

Note that Figure 1 and all the flow charts in this paper make use of the following PL/1 notations:

1. A.B is used to denote the quantity B contained in A. For example, PTW.ADDR denotes the ADDR field within the PTW.

2. The double vertical bar (||) is used to denote concatenation (e.g., PTW. ADDR || 000000).

3. The single vertical bar (|) is used to denote the logical inclusive OR.

Descriptor Segment Base Register

The Descriptor Segment Base Register (DSBR) is an internal processor register used to locate the current descriptor segment. In the new processor, the DSBR has been extended to 51 bits to accommodate the longer address and bound fields and to contain a stack offset. The DSBR is loaded from and stored into a doubleword having the same format as a Segment Descriptor Word (SDW) with the exception that unused fields are ignored during loading of the DSBR and are set to zero when the DSBR is stored. Only the following four SDW fields have meaning when loaded into a DSBR.

1. ADDR (24 bits)

2. BOUND (14 bits)

3. U (paged/unpaged switch; 1 bit)

4. STACK (12 bits)

The STACK field specifies the upper 12 bits of the 15-bit stack segment number. This register is used only during the execution of a CALL instruction.

RINGS AND RING BRACKETS

A Multics process consists of procedure and data segments which are all directly addressable through the descriptor segment of that process. However, a process may access a segment only when the process is running at an appropriate level of privilege. For example, all the segments of the

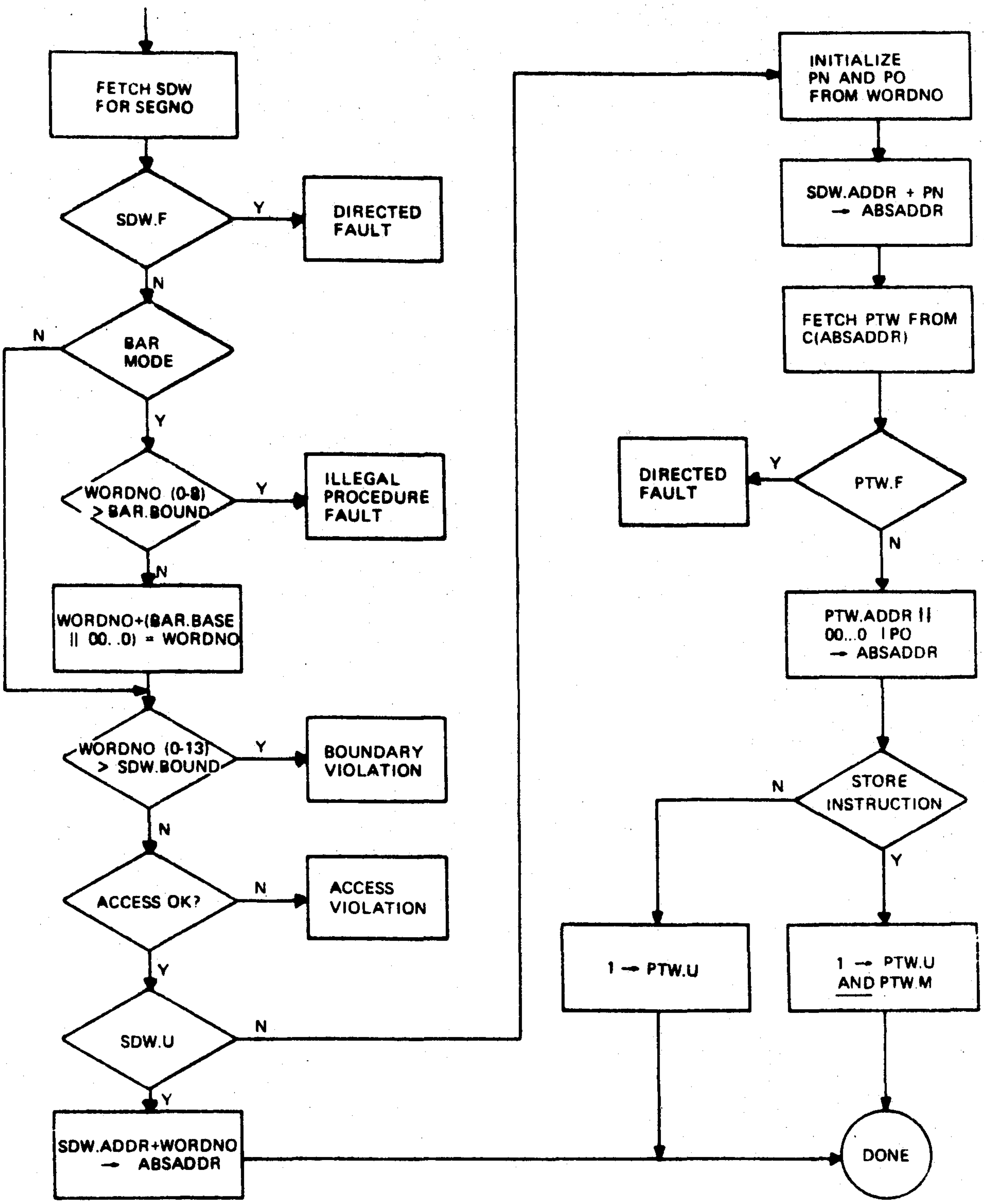


Figure 1. Appending Cycle

hardcore supervisor are shared and accessible to all Multics processes but only when executing at the highest level of privilege.

The Multics system allows segments to be grouped into an ordered set of collections called rings in which segments requiring the highest level of privilege to reference can be accessed only from within the innermost ring of the set. Each ring is identified with a ring number designating the required level of privilege necessary to access segments in that ring. In Multics, the ring with the highest privilege is ring 0, which contains the procedures and data bases of the hardcore supervisor. Each user process has at least two rings, one for the hardcore supervisor and one for user programs and data. The user process may generate more rings (levels of lesser privilege) if desired.

Frequently, it is useful to allow a segment to be accessible in more than one ring. For example, it is often useful for a data base which is writeable in an inner ring to be readable in an outer ring. For this reason, the concept of ring brackets was introduced.

The access of a user to a specific segment is controlled by two quantities: the access attributes (e.g., read, execute, write) and the ring brackets. The ring brackets of a segment are specified by three integers (R1, R2, and R3) each of which must be greater than or equal to the preceding number. The first number (R1) specifies the top (highest ring number) of the read/write bracket, the second number (R2) specifies the top of the read/execute bracket, and the last number (R3) specifies the top of the call bracket.

Read/Write Bracket (Rings 0-R1)

Attempts to read or write a segment by a procedure executing in a ring within the read/write bracket are allowed if the appropriate (read or write) access indicators are on for the segment being referenced. Execution of a procedure in a ring within this bracket is permitted only at the top of the read/write bracket (R1), which is also the bottom of the read/execute bracket. Note that the highest ring from which a segment can be written is specified by R1. As a result, the data in the segment is no more reliable than the procedure segments which operate in that ring.

Read/Execute Bracket (Rings R1-R2)

Attempts to read or execute (transfer to) a segment by a procedure executing in a ring within the read/execute bracket are allowed if the appropriate (read and execute) indicators are on for the segment being referenced. Writing of a segment within its read/execute bracket is permitted only from the ring at the bottom of the bracket (R1), which is also the top of the read/write bracket. If R2 is equal to R1, the read/execute bracket specifies a single ring (R1).

Call Bracket (Rings (R2 + 1) - R3)

Attempts to call a (procedure) segment from a segment executing in a ring above the read/execute bracket but within the call bracket of the procedure to be called are allowed if the execute indicator of the procedure to be called is on and if the new CALL instruction (described below) is used. When the CALL instruction is directed to a procedure in an inner ring which has the appropriate execute access and call bracket, the processor atuomatically switches to the ring specified as the R2 of the procedure being called. The call bracket and the CALL instruction are the only means (except for faults) by which control can be passed from an outer ring to an inner (more privileged) ring. If R3 is equal to R2, the call bracket is null and the procedure cannot be called from an outer ring.

Summary

Assuming that the appropriate (read, execute, or write) indicators are on, the following list summarizes the effects of the three ring brackets:

1. Writing is permitted from a ring within the read/write brackets only (i.e., if ring $\le$ R1).

2. Reading is permitted from a ring within the read/write bracket or the read/execute bracket (i.e., ring $\le$ R2).

3. Execution (or transfer of control) is permitted only from a ring within the read/execute bracket (i.e., R1 $\le$ ring $\le$ R2).

4. Calling (via CALL only) is permitted from a ring within the read/execute or call brackets (i.e., R1 $\le$ ring $\le$ R3).

5. The CALL instruction is the only instruction which may be used to access a segment in a ring within its call bracket (i.e., R2 < ring $\le$ R3).

6. No access is permitted to a segment from a ring higher than the call bracket (i.e., ring > R3).

PROCESSOR ADDRESS REGISTERS

Like the 645, the new processor has 10 address or pointer registers (PRs). Eight of these pointer registers can be directly accessed and modified by the software, one is used to locate the current instruction, and one is used exclusively by the processor for effective address calculations. Unlike the 645, each of the eight program addressable pointer registers specifies a full segmented address including the segment number and the word number in a single pointer register. These registers have also been extended to include a bit number.

Instruction Pointer Register

The instruction pointer register (IPR) is used by the processor to locate the current instruction and may be modified by the software to effect a transfer of control. The IPR is actually an extension of the PBR and IC of the 645. The contents of the 36-bit IPR are outlined below.

PRR	PSR	IC

PRR (3 bits) Is the procedure ring register and specifies the ring (level of privilege) in which the processor is currently executing. PRR may be set to a higher value only by an RTCD or RCU instruction. It may be set to a lower value only by a CALL instruction (see below) or by a fault or interrupt.

PSR (15 bits) Is the procedure segment register (same as the PBR in the 645) and specifies the segment number of the current procedure segment.

IC (18 bits) Is the instruction counter (same as in the 645).

Temporary Pointer Register

The temporary pointer register (TPR) is used exclusively by the processor for operand address calculations and serves the same general purpose as the TBR and computed address (CA) of the 645.

TRR	TSR	CA	BITNO

TRR (3 bits) Is the temporary ring register and is used to maintain the lowest level of privilege (i.e., highest ring number) encountered during operand address calculation. The TRR is initialized with the value of the PRR field of the IPR at the beginning of each instruction. During the operand address calculation, TRR is used to record the highest value of SDW.R1 (the top of the read/write bracket) of any segment used, in the address calculation. For example, if an indirect address is fetched from segment X, TRR is set to the larger of TRR (its current value) and the R1 field in the SDW for segment X. Note that during the operand address calculation, the value of TRR may get larger but never smaller.

TSR (15 bits)	Is the temporary segment register (same as the TBR in the 645) and is initialized with the value of the PSR field of the IPR at the beginning of each instruction. During operand address calculation, TSR contains the segment number portion of the current address calculation.
CA (18 bits)	Is the computed address and serves the same function as the 645 register of the same name. The computed address is initialized at the beginning of each instruction with the contents of the instruction counter of the IPR. During operand address calculation, CA contains the word number portion of the current address calculation.
BITNO (6 bits)	Is a bit-offset relative to the first bit in the word specified by CA. This field is ignored by all instructions except the new instructions specifically designed for string manipulation or decimal arithmetic.

Once an operand address calculation is complete, the value of TRR is compared with the ring brackets of the segment containing the operand address to determine whether the operation is to be allowed. For example, if the instruction intends to store into this operand, the value of TRR must be less than or equal to the R1 (in the SDW) of the segment to be modified.

Eight Pointer Registers

The new processor contains eight program accessible pointer registers, which replace the eight address base registers (ABRs) of the 645. The PRs of the new processor differ from the ABRs in that each PR contains both a segment number and a word number portion. In effect, each PR of the new processor behaves as a 645 base register pair. The contents of each 42-bit PR are outlined below.

RN	SEGNO	WORDNO	BITNO

RN (3 bits) — Is used by the software to specify the level of privilege (i.e., ring number) at which the processor is to treat the address contained in the address register. When the processor uses the contents of a PR for address modification (e.g., bit 29 in the instruction word is on) the value of TRR is set to the larger of TRR (its current value) and the RN field of the specified PR. The use of the RN field of a PR allows the software to save the TRR of an operand address calculation — e.g., through the use of an EAP (effective address to pointer) instruction.

SEGNO (15 bits) — Specifies the segment number portion of the segmented address.

WORDNO (18 bits) — Specifies the word number portion of the segmented address.

BITNO (6 bits) — Specifies a bit-offset relative to WORDNO and is ignored by all instructions except those designed specifically for string manipulation or decimal arithmetic.

The software may store the contents of a PR into an ITS (indirect to segment) word pair with the use of an STP (store pointer) instruction. The software may then address indirectly through the ITS indirect word rather than using the original PR. Alternatively, the software may reload another PR from the ITS word pair through the use of the EAP instruction. In either case, it is necessary to save the value of the RN of the PR in the ITS word pair so that the privilege level of the original operand address calculation is not lost. As a result, the ITS word pair is modified to include a ring number field as outlined below.

	SEGNO	RN		ITS	
WORDNO		BITNO		MOD	

SEGNO (3-17) — Is the segment number field (as in the 645). Note that bits 0-2 of the ITS word pair are set to zero for compatibility with 645 programs expecting an 18-bit segment number in the upper half of the first word.

RN (18-20) — Is set to the value of the RN field of the PR during the STP instruction. If the processor attempts to indirect through an ITS word pair, TRR is set to the larger of TRR, RN (of the ITS), and R1 of the segment containing the ITS. Note that an improper value of RN in an ITS word pair has no ill effect, since the processor always takes the maximum of TRR and RN. In other words, it is impossible for an ITS word pair to specify a higher privilege than the segment in which it resides.

ITS (30-35)	Specifies the modifier code (octal 43) for the ITS modifier (same as in the 645).
WORDNO (0-17)	Is the word number portion of the saved PR.
BITNO (21-26)	Is the bit-offset of the PR saved by the STP instruction. The strange placement of the BITNO field is necessary to remain compatible with the current PL/1 software implementation of bit-offsets.
MOD (30-35)	Is set to zero by the STP instruction but may be set by the software to specify further address modification (same as in the 645).

Since most Multics compilers (notably PL/1) calculate addresses via an EAP instruction, it can be expected that compiler generated code can take full advantage of the hardware protection mechanism with little modification. If all addresses of all input parameters are calculated and saved (for use as outgoing argument pointers) via the use of the EAP and STP instructions, it will be possible for a procedure operating in ring 1 to pass to ring 0 a parameter given to the procedure from ring 2, without checking the address of the parameter. The access checking is fully automatic as long as the TRR of the original address calculation continues to be maintained and passed along as the RN field of a PR or ITS word pair.

The STCD (store control double) instruction is modified to store the PRR in the same manner as STP stores the RN field of a PR. The PRR is stored by the STCD to allow an RTCD (return control double) instruction to return to the proper ring.

ACCESS CONTROL MECHANISM

Figures 2 through 6 attempt to flow chart the entire access control mechanism from the instruction fetch up to actual execution of the instruction. In order to concentrate on the access control mechanism, many details have been left out of the flow charts (indexing, IT modifiers, etc.). If all the access control checks are successfully met, control will end up in a circle marked "done." The contents of the flow charts are summarized below.

Figure 2 begins with the instruction fetch and continues through the initial address calculation.

Figure 3 shows how indirect addressing affects the access computation. (The notation "R1 (ITS)" is used to denote the R1 of the segment containing the ITS word pair.)

Figure 4 shows the access checks made for all instructions except for transfer of control.

Figure 5 shows the access checks performed on all transfer instructions with the exception of the CALL instruction. Note that the PRR cannot be changed by a normal transfer instruction (even to a higher value). However, it is possible to set the PRR to a higher value with the modified RTCD (described below).

Figure 6 shows the access checks performed by the CALL instruction (the only slave instruction permitted to set the PRR to a lower value).

CALL INSTRUCTION

The CALL instruction is provided as the only means by which a procedure segment may call a procedure in an inner ring (i.e., set PRR to a lower value). The CALL instruction is to be used in all standard interprocedure calls and is intended to replace the transfer instruction as the last instruction of the standard Multics calling sequence.

The CALL instruction uses two PRs: PRn and PRn+1, where n is even. The value of n is wired into the processor and is currently 6. It is possible for a field engineer to change this value to 4, 2, or 0 by an orderly procedure. It must not be possible, however, to change this value under user program control. (This use of a pair of PRs involves two full PRs (RN, SEGNO, WORDNO, and BITNO) and should not be confused with a 645 base register pair.) When the CALL instruction is used to transfer control to another ring, the assumption is made (by convention) that the stack segment of the target ring has a segment number equal to the ring number of the target ring (i.e., the stack segment for ring X is a segment number X). The CALL instruction behaves as a TRA (transfer) instruction with the following exceptions:

1. The access checking for a CALL instruction allows PRR to be set to a lower value, provided that the call is made from a ring within the call bracket of the target segment (see Figure 6).

2. If an attempt is made to call a procedure in an outer ring (a relatively rare case), an access violation occurs. Because of the necessity of copying all arguments, the standard call, save, and return sequences cannot handle calls to an outer ring without excessive software overhead. Therefore, calls to outer ring procedures will continue to cause a fault to allow the system software to interpret the call.

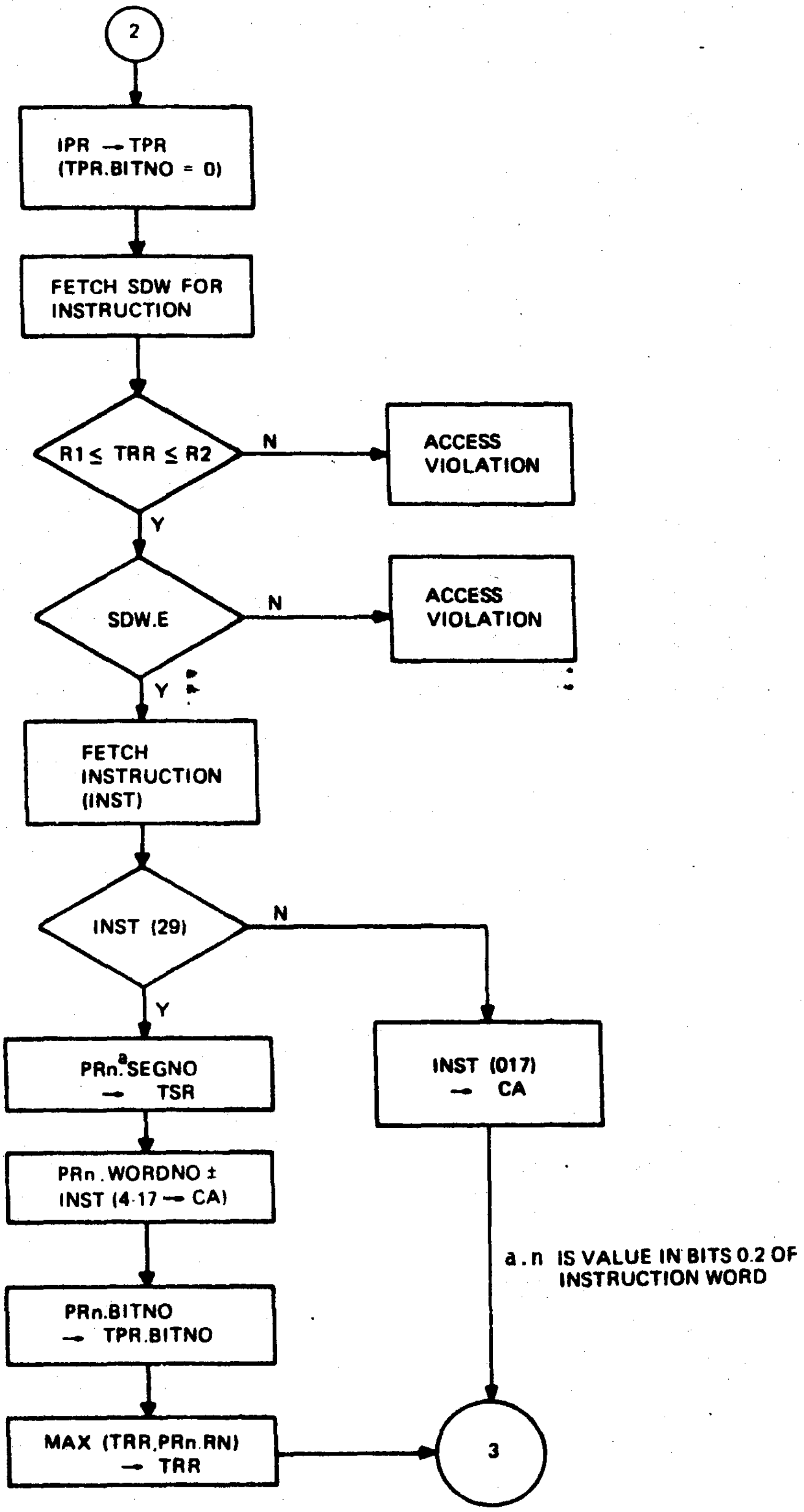


Figure 2. Instruction Fetch and Initial Address Calculation

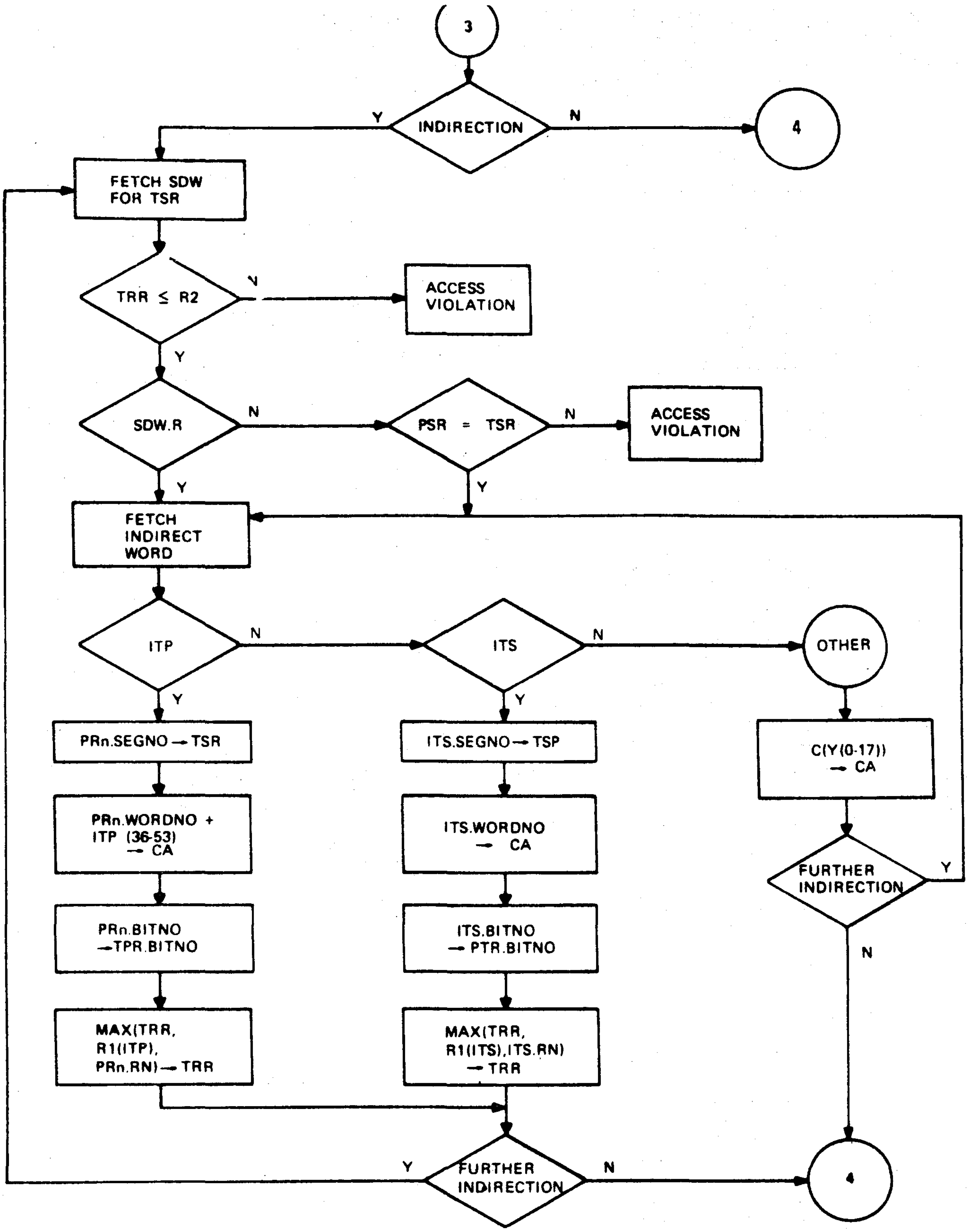


Figure 3. Indirect Addressing

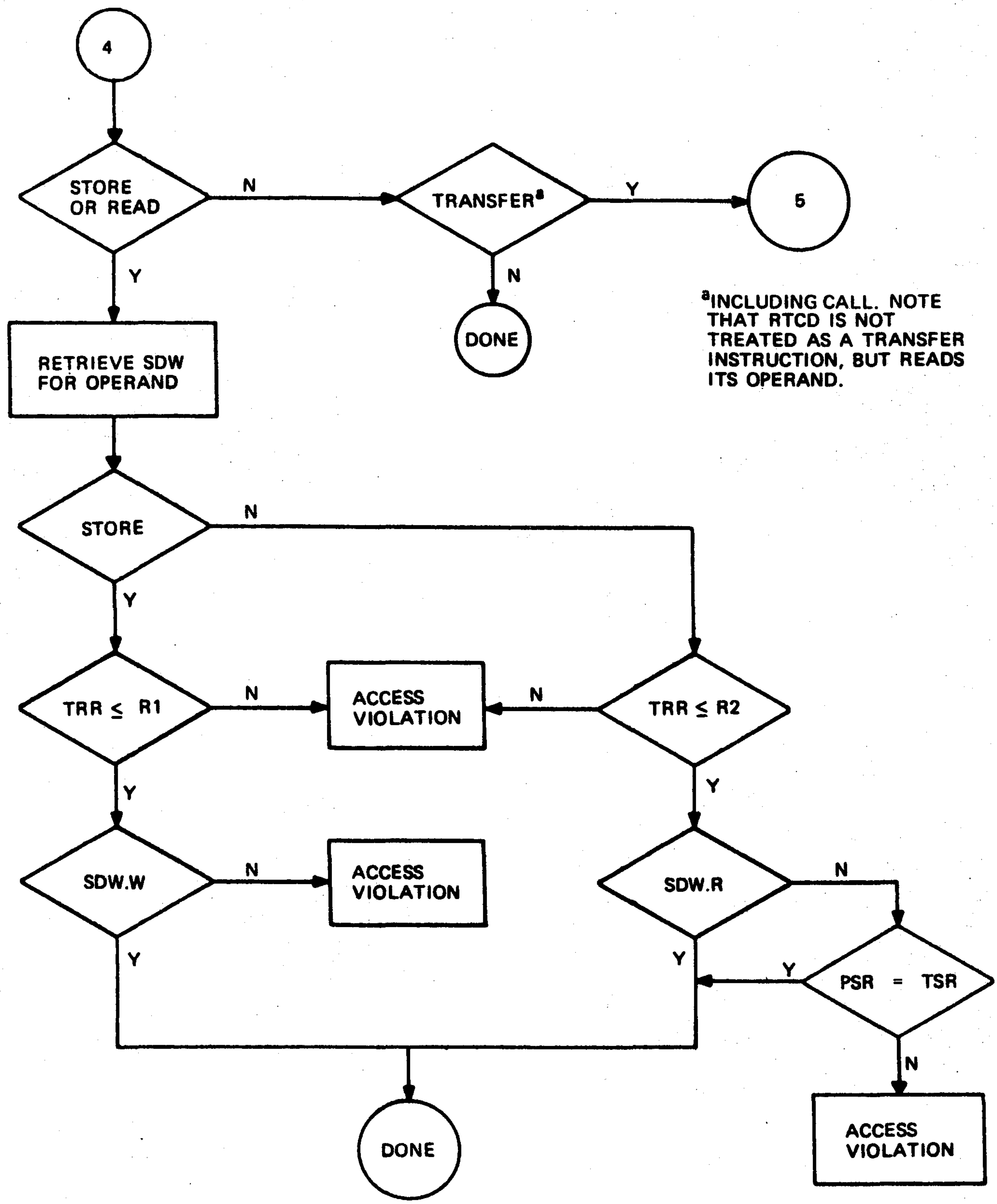


Figure 4. Access Checks for Nontransfer Instructions

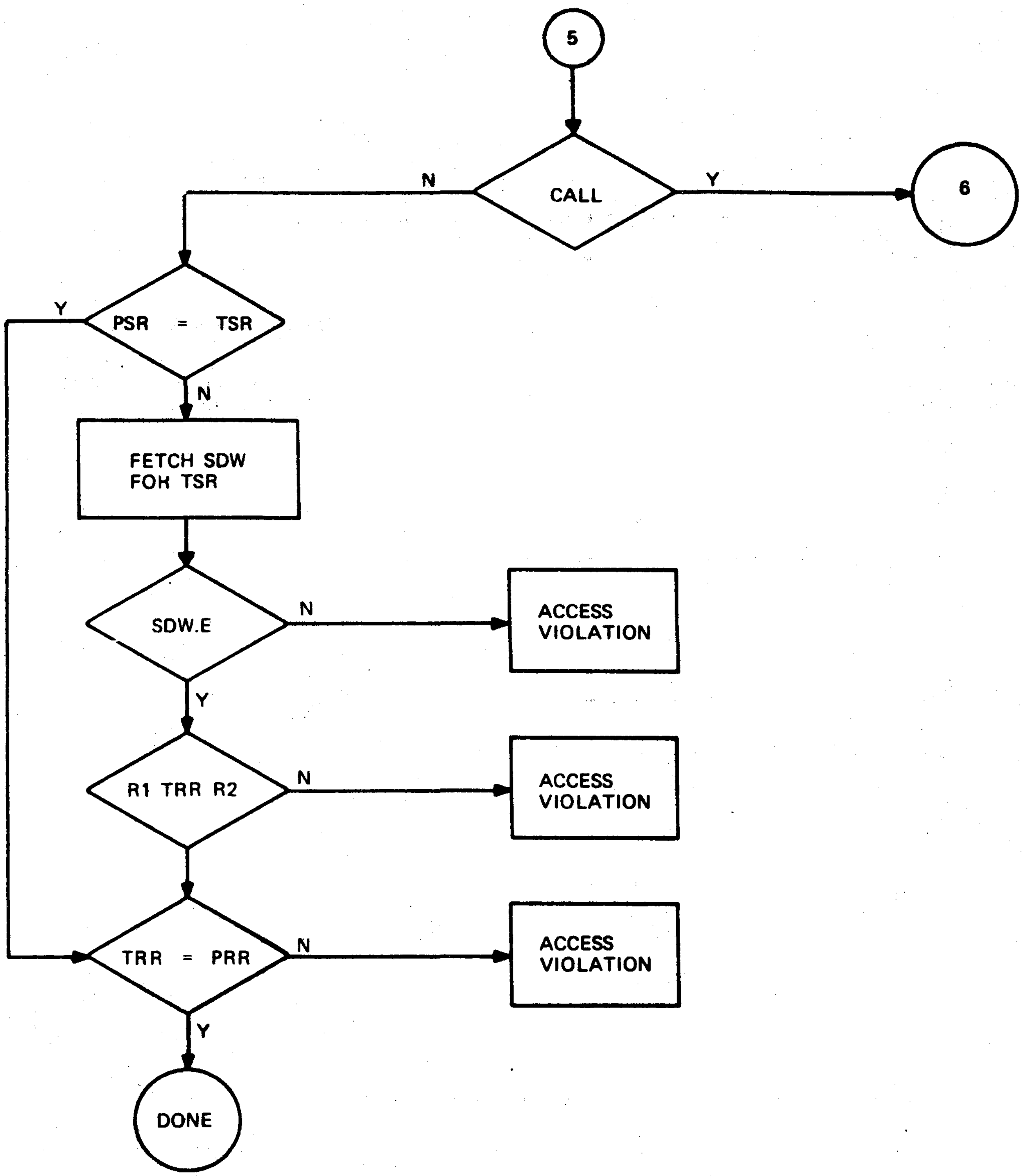


Figure 5. Access Checks for Transfer Instructions, Except CALLn

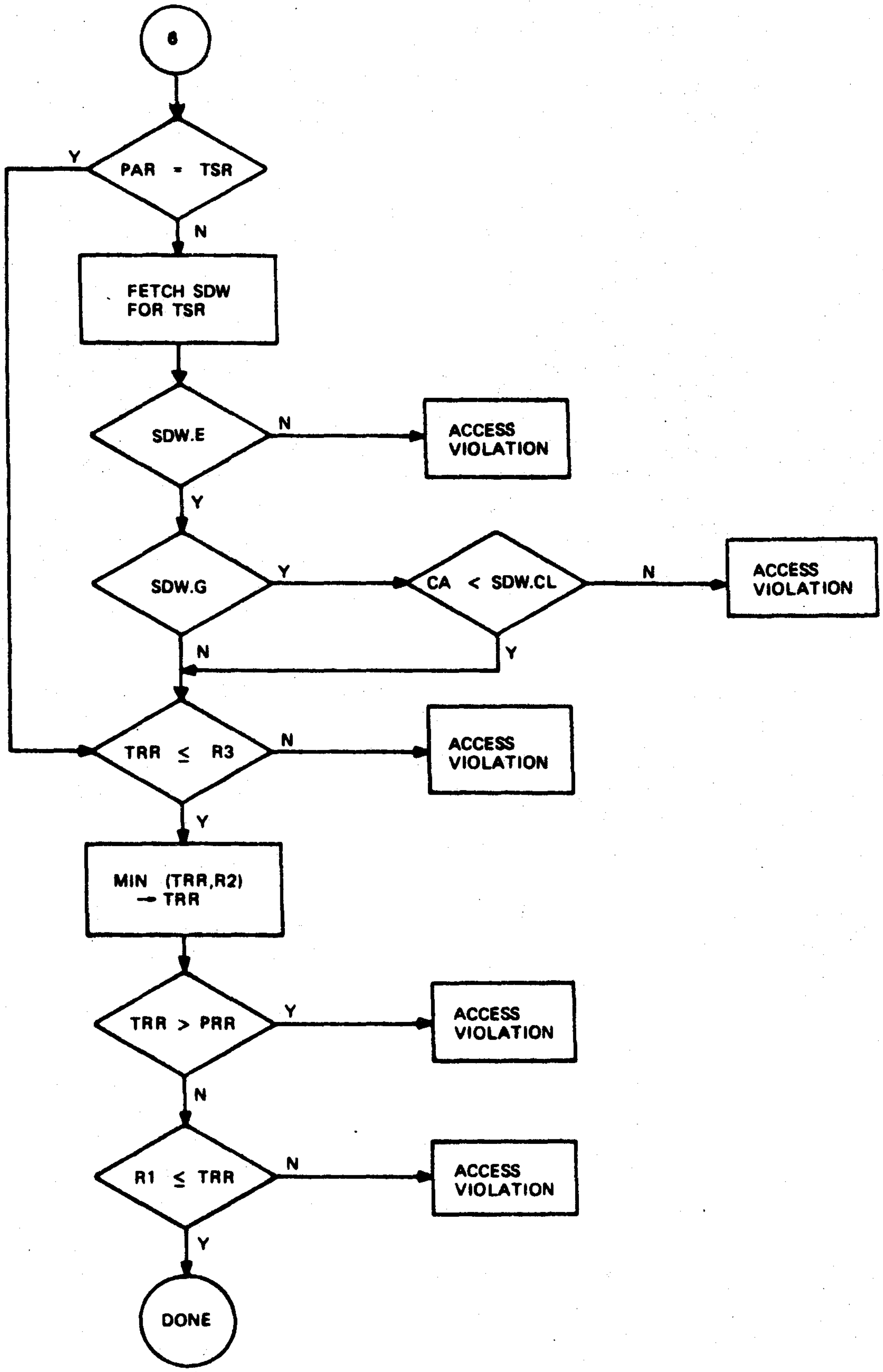


Figure 6. Access Checks for CALL Instruction

3. At the beginning of the CALL, the contents of PRn are assumed to point to a location (i.e., the beginning of the current stack frame) within the stack segment of the calling ring. During the execution of the CALL, the processor sets the contents of PRn+1 to point to word 0 of the stack segment of the target ring in one of two ways:

   a. If control is to remain in the current ring (i.e., TRR = PRR), the SEGNO portion of PRn+1 is set to the SEGNO of PRn and the WORDNO portion of PRn+1 is set to zero.

   b. If control is to be passed to an inner ring (i.e., TRR < PRR), the SEGNO portion of PRn+1 is set to the value of the target ring number (TRR) and the WORDNO of PRn+1 is set to zero.

If an attempt is made to call to an outer ring (i.e., TRR > PRR), an access violation is generated as indicated in Figure 6.

Figure 7 details the operation of the CALL instruction after the effective address computation has been completed (i.e., TPR has been computed), and TRR is set to the target ring number (see Figure 6).

The software stores a pointer to the end of the current (or last used) stack frame in the beginning of that stack. The standard call and save sequences might then be modified as follows:

Calling Sequence:	ZERO	ARGLIST	ZERO points to ARGLIST
	STCD	6 | 20	Set return location
	CALL	ENTRYPOINT	Call external procedure
Save Sequence:	EAP1	7 | NEXT,*	Load pointer with base of new stack frame
	STP6	1 | 16	Save pointer to old frame
	EAP6	1 | 0	Switch to new frame
	EAP1	6 | TEMP	Compute pointer to next frame (allocate new frame)
	STP1	6 | 18	Save pointer to next frame
	STP1	7 | NEXT	Update stack base
	STP0	6 | 26	Save ARGLIST PR

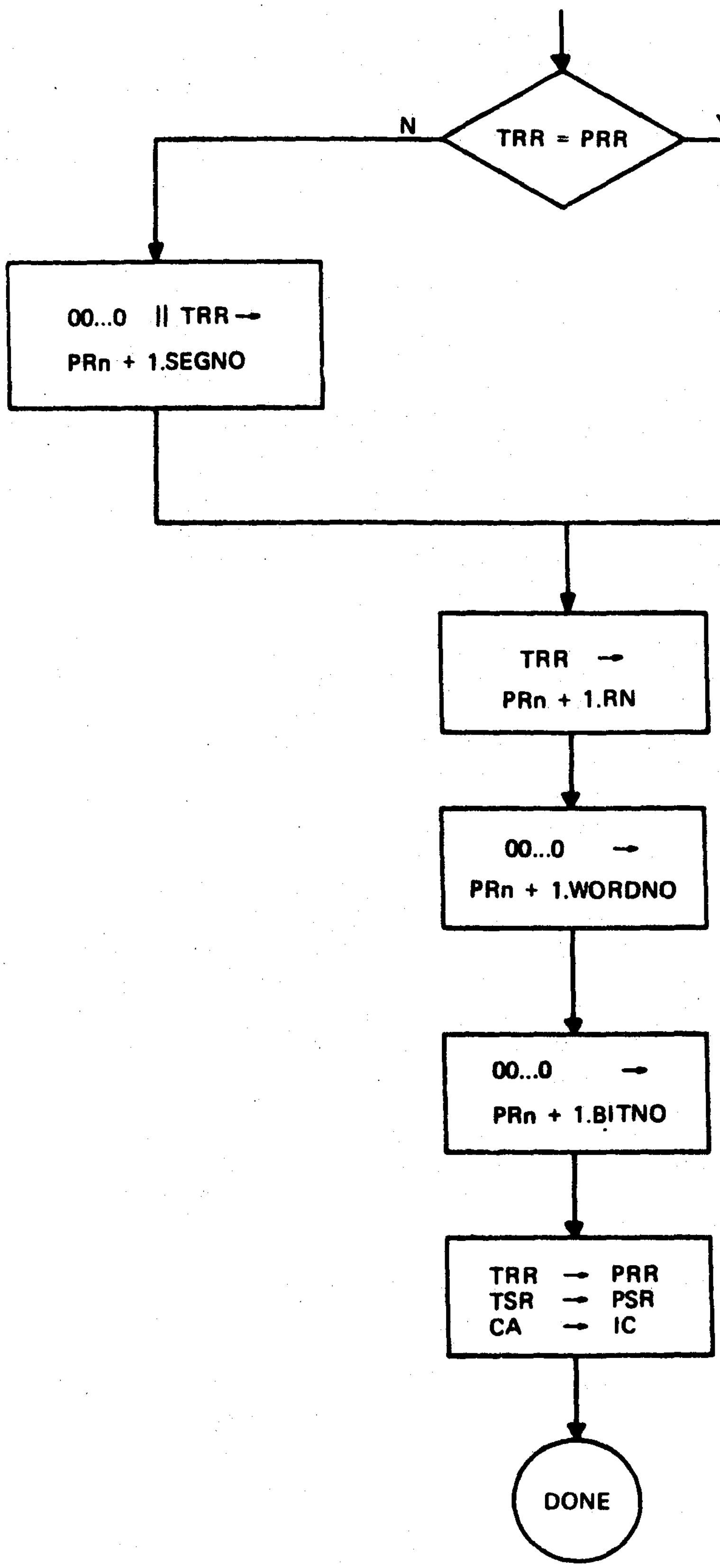


Figure 7. Execution of CALL Instruction

ASSOCIATIVE MEMORY

As in the 645, the new processor requires a small associative memory in order to avoid memory fetches of frequently used SDWs and PTWs. A series of measurements and experiments has determined the effectiveness and behavior of the 645 associative memory. The experiments were conducted and measurements taken during normal Multics operation, under varying user loads.

The experiments indicate that the current 645 associative memory is quite effective. It appears that the most significant aspect of the associative memory is in the speed of the search, since it is not possible to overlap completely the associative lookup with other work. This aspect suggests that a one-pass lookup would be a desirable objective. There are at least three ways in which the effect of a one-pass lookup can be achieved:

1. One approach is derived from the fact that the "hit rate" on SDWs for paged segments on the 645 is extremely low (about 0.21 percent). This fact suggests a one-pass search of an associative memory containing only PTWs and SDWs for unpaged segments. The search would look for an unpaged SDW for the referenced page within the segment. The copy of the PTW in the associative memory must be extended to include access control information from the original SDW for the segment. This approach has the drawback that any change in the operating environment (e.g., the use of smaller page sizes) which causes SDWs for paged segments to be in higher demand would begin to degrade system performance.

2. Another approach is to achieve the effect of a one-pass search using a two-pass search and overlapping the first pass during address preparation. In this approach, the single associative memory contains both SDWs (paged and unpaged) and PTWs extended with access control information. During address preparation, the associative memory is searched for the SDW of the segment to be referenced. After address preparation, a second pass is made to locate the PTW for the page to be referenced. Only if the second pass fails are the results of the first pass interrogated. If the first pass had succeeded, only the PTW must be fetched from core memory. Otherwise, both the SDW and the PTW must be fetched from core memory.

3. A third approach is to search two associative memories in parallel, one for SDWs and the other for PTWs. If either the SDW or PTW is not found in its respective associative memory, it is retrieved from core memory and updated into the appropriate associative memory. Although this approach requires duplicate circuitry, it is appealing in its logical simplicity and is the method chosen.

D. Abbreviations and Acronyms

ADDR	Address portion of PTW
AST	Active Segment Table
ASTE	Active Segment Table Entry
CA	Computed Address
CM	Core Map
CME	Core Map Entry
DBR	Descriptor Base Register
DC	Directory Control
DCW	Data Control Word
DID	Device Identifier
DIM	Device Interface Module
DS	Descriptor Segment
DSBR	Descriptor Segment Base Register
IC	Instruction Counter
IPR	Instruction Pointer Register
ITP	Indirect to Pointer Register
ITS	Indirect To Segment
KST	Known Segment Table
KSTE	Known Segment Table Entry
MC	Memory Controller
PC	Page Control
PHM	Page Has Been Modified
PHU	Page Has Been Used
PN	Page Number
PO	Page Offset

PR	Pointer Register
PRR	Procedure Ring Register
PSR	Procedure Segment Register
PT	Page Table
PTW	Page Table Word
RA	Ring Alarm register
RN	Ring Number
SC	Segment Control
SDW	Segment Descriptor Word
SFH	Segment Fault Handler
SST	Systems Segment Table
TPR	Temporary Pointer Register
TRR	Temporary Ring Register
TSR	Temporary Segment Register
UID	Unique Identifier