Honeywell

THE MULTICS VIRTUAL MEMORY

MULTICS

SUBJECT:

Technical Papers on the Multics Virtual Memory as Designed for Model 645 Implementation.

SPECIAL INSTRUCTIONS:

These technical papers are reprinted here for theoretical and historical purposes only.

DATE:

June 1972

ORDER NUMBER:

AG95, Rev. 0

PREFACE

This document consists of three technical papers which describe the theory and practice of Multics virtual memory implementation. Multics (Multiplexed Information and Computing Service) is a general purpose computer system which has been designed to be a "computer utility." As such, it is essential that Multics provide its users with sufficient resources to do a wide variety of tasks and that the system be protected from destructive interactions between users. The papers address the theory, the practice, the hardware, and the software used to provide an effectively infinite memory to each user and to protect both the users and the system.

The first paper discusses the concept of a virtual memory and explores several ways in which such a memory could be implemented. The method used to implement the Multics virtual memory on the Series 600 Model 645 processor is developed in detail. This paper is of historical importance and presents valid Multics design theory although the Model 645 is no longer used as the Multics processor.

The second paper extends the discussion further into the subject of protection. The theory of the Multics ring structure is introduced and its implementation on the Model 645 is described. This theory is still valid although the Model 645 is no longer used as the Multics processor.

The third paper shows how the features described in the two earlier papers are handled by hardware under the optional Multics modifications to Series 6000 processors. Several new processor features are introduced and described. Use of these features allows Multics to run on the Series 6000 processors, specifically the Model 6180, with greatly increased efficiency as compared with the earlier implementation of Multics on the Series 600 processors.

File No.: 1LW3

An alphabetized list of abbreviations and acronyms used in all three papers has been included as an aid to the reader.

Papers in this document were written to further the understanding of the Multics design philosophy and practices. They are not intended to be specifications of the Multics system or its components. Authors have made simplifying assumptions at times to make the main point clearer and easier to understand. Persons requiring design specification details are requested to contact the Multics development staff for guidance and assistance.

"The Multics Virtual Memory" was first published as Technical Information Series Report R69LSD3, Copyright 1970 by General Electric Company, U.S.A.

"Access Control to the Multics Virtual Memory" was first published as Technical Information Series Report R69LSD4, Copyright 1970 by General Electric Company, U.S.A.

CONTENTS

A. The Multics Virtual Memory

CONTENTS

PREFACE

In the past few years several well-known systems have implemented large virtual memories which permit the execution of programs exceeding the size of available core memory. These implementations have been achieved by demand paging in the Atlas computer, allowing a program to be divided physically into pages only some of which need reside in core storage at any one time, by segmentation in the B5000 computer allowing a program to be divided logically into segments, only some of which need be in core, and by a combination of both segmentation and paging in the 645 and the IBM 360/67 for which only a few pages of a few segments need be available in core while a program is running.

As experience has been gained with remote-access, multiprogrammed systems, however, it has become apparent that, in addition to being able to take advantage of the direct addressibility of large amounts of information made possible by large virtual memories, many applications also require the rapid but controlled sharing of information stored on-line at the central facility. In Multics (Multiplexed Information and Computing Service), segmentation provides a generalized basis for the direct accessing and sharing of on-line information by satisfying two design goals: 1) it must be possible for all on-line information stored in the system to be addressed directly by a processor and hence referenced directly by any computation. 2) it must be possible to control access, at each reference, to all on-line information in the system.

The fundamental advantage of direct addressability is that information copying is no longer mandatory. Since all instructions and data items in the system are processor-addressible, duplication of procedures and data is unnecessary. This means, for example, that core images of programs need not be prepared by loading and binding together copies of procedures before execution; instead, the original procedures may be used directly in a computation. Also, partial copies of data files need not be read, via requests to an I/O system, into core buffers for subsequent use and then returned, by means of another I/O request, to their original locations; instead the central processor executing a computation can directly address just those required data items in the original version of the file. This kind of access to information promises a very attractive reduction in program complexity for the programmer.

If all on-line information in the system may be addressed directly by any computation, it becomes imperative to be able to limit or control access to this information both for the self-protection of a computation from its own mishaps, and for the mutual protection of computations using the same system hardware facilities. Thus it becomes desirable to compartmentalize or package all information in a directly-addressible memory and to attach to these information packages access attributes describing the fashion in which each user may reference the contained data and procedures. Since all such information is processor-addressible, the access attributes of the referencing user must be enforced upon each processor reference to any information package.

Given the ability to directly address all on-line information in the system, thereby eliminating the need for copying data and procedures, and given the ability to control access to this information, then controlled information sharing among several computations follows as a natural consequence.

In Multics, segments are packages of information which are directly addressed and which are accessed in a controlled fashion. Associated with each segment is a set of access attributes for each user who may access the segment. These attributes are checked by hardware upon each segment reference by any user. Furthermore all on-line information in a Multics installation can be directly referenced as segments while in other systems most on-line information is referenced as files.

Chapter 1

GENERAL PROPERTIES OF THE MULTICS VIRTUAL MEMORY

1. INTRODUCTION

In recent literature the term "virtual memory" has become quite familiar. The adjective "virtual" suggests that this memory is the image of an ideal memory that one would like to have, since it complies with the actual needs of a multiprogramming, multiple-access computer utility. This "ideal memory" is not available as a hardware device and has been simulated by the Multics system using a conventional memory with the assistance of additional hardware and software features.

This chapter describes the properties of the ideal memory, justifies the desire for these properties, and explains the principles of the simulation of this memory.

2. THE IDEAL MEMORY

In order to describe this ideal memory the terms "segment" and "segmented memory" need to be defined first.

2.1. Segments

A segment is an entity defined by:

1) A name which uniquely identifies the segment.
2) A descriptor which describes the properties or "attributes" associated with the segment.
3) A body which is an array of consecutive elements.

The name is a character string of arbitrary length.

The descriptor contains all attributes the system designer needs to attach to the segment: the size and the physical location of the body, access rights for different users with respect to this segment, the date it was created, etc.

The body of the segment is an ordered set of elements, called words, each of which is identified within the segment body by an integer i, its index. The number of elements in the body is called the length of the segment.

2.2. Segmented Memory

A segmented memory will be defined as a memory with the following properties:

1) It is capable of containing segments and only segments.
2) If it contains a segment named n, then n is the address of the descriptor of this segment and the pair [n,i] is the address of the ith element in the body of this segment.

3) It is capable of performing operations on the descriptor and the body of any segment, in accordance with the attributes recorded in the descriptor.

2.3. Ideal Memory

The ideal memory can now be defined as a large, segmented memory directly accessible by the processor, where by "large" it is meant that the maximum number of segments that one can store in it is adequate for the needs of the system.

A simple representation of such a memory is shown in Figure 1; it comprises a memory controller (MC), a large number of descriptors each of which contains the name and the attributes of a segment, and a large number of linear memories each of which is connected to a descriptor and can contain the body of a segment.

The processor can send two types of requests to the MC: requests for operations on descriptors and requests for operations on bodies. In both cases the processor must communicate to the MC the identification of the user on behalf of whom the operation is requested.

2.4. Operations on Descriptors

The general form of a request sent by the processor to the MC for operations on descriptors is

OPCODE n arguments userid

where:

- OPCODE designates operations, such as "create a segment", "change the length of a segment", "change access rights";

- n is the name of the segment. The MC uses it to locate the appropriate segment descriptor;

- arguments are parameters associated with the function defined by OPCODE;

- userid is the identification of the user on behalf of whom the operation is requested. The MC uses this userid in order to determine from the attributes of the segments whether this particular user has the right to perform this particular operation.

2.5. Operations on Segment Bodies

The general form of a request sent by the processor to the MC for operations on segment bodies is

OPCODE [n,i] userid

where:

- OPCODE designates operations, such as "read", "write", "instruction fetch";

- n is the name of the segment; the MC uses it to locate the appropriate segment descriptor. It then uses the segment descriptor to locate the segment body;

- i is the index of the word within the segment to which the operation is to be applied;

- userid is used by the MC as above.

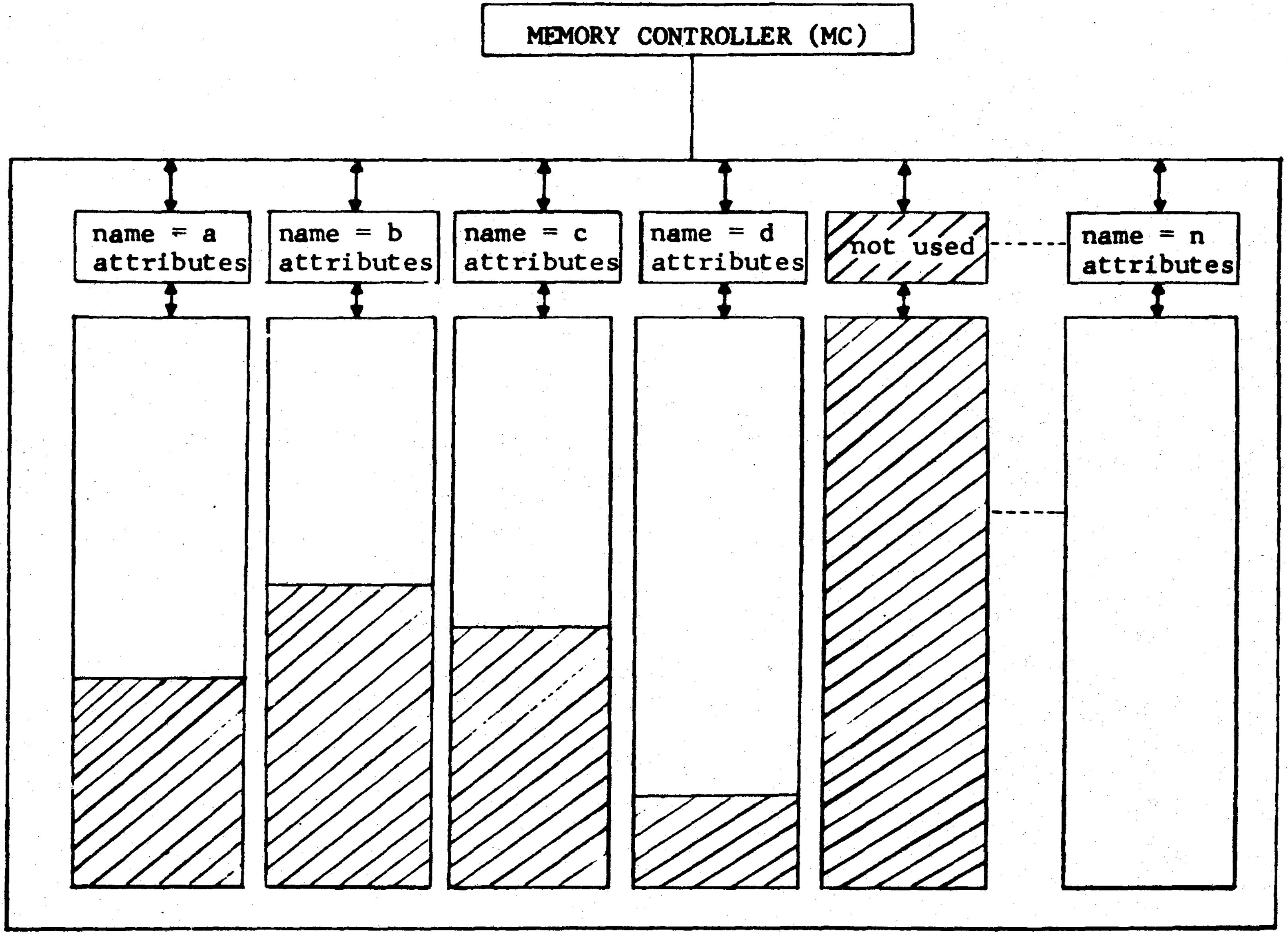


Figure 1. Organization of the Ideal Memory

3. JUSTIFICATION OF THE IDEAL MEMORY PROPERTIES

The ideal memory has been defined as a "large segmented memory directly accessible by the processor". The advantages of such an ideal memory will be explained by successively introducing the advantages of a memory, that is: 1) large (but not segmented); 2) segmented (but not large); and finally, 3) large and segmented.

3.1. Large, Unsegmented Memory

Because the memory is large and directly accessible by the processor, the user is provided with a core memory large enough for any of his computations. Therefore, he can run a program without being concerned with its size. However, no matter how large the core memory is, if it is a linear memory accessible by a single number, no sharing of information in core can be tolerated between programs of different users since no protection mechanism is in effect at the time a word is accessed.

3.2. Small, Segmented Memory

Because the memory is segmented and directly accessible by the processor, the user is provided with several independent linear core memories in each of which he can store one of his segments, deciding who can access it and how. Therefore, the same segment can be shared in core by several user programs without the danger of unauthorized accessed to this segment. However, even though the memory is segmented, if the number of segments that one can store in it is small, the user is faced with the problem of overlays.

3.3. Large, Segmented Memory

By having the two properties "large" and "segmented" a directly accessible memory provides the user with:

- a large machine-independent memory. There is a one-to-one correspondence between the name by which the user references a one-word datum and the physical location in memory where the datum resides. As a consequence, users are provided with a simple means of writing programs such that, when executed, they access common information in core. They merely have to reference this information by its name.

- a protection mechanism. This mechanism is in effect during execution at any memory access and protects segments from unauthorized access.

3.4. Note on Information Sharing

It is worth making some remarks about information sharing. Information to be shared consists of data and procedures.

- Sharing data or procedures in core requires:

  a) A mechanism by which a reference to a segment by its name X will cause segment X, and not a copy segment X, to be referenced during program execution.

  b) A mechanism by which the shared information can be protected from unauthorized access while it is in core.

- Sharing procedures in core also requires:

  c) A mechanism by which one can produce pure procedures that can be executed simultaneously by several programs.

The memory described here provides a) and b), but not c). In fact the memory itself cannot provide c); writing a pure procedure implies the ability of communicating as parameters to this procedure the names of any information private to the program on behalf of which the procedure is executing. These names cannot be stored in the memory itself; they have to be stored in processor registers whose names are invariant. During execution of a pure procedure by a processor on behalf of a program, the names of data segments private to the program are stored in processor registers whose names are stored in the pure procedure. The processor requests the data from the memory controller using the name found in the appropriate processor register.

4. PRINCIPLE OF THE SIMULATION

The memory presented here is simulated in the Multics system, this simulation being achieved by a combination of hardware and software features. Hardware segmentation has been implemented in the 645 and constitutes the most important of the hardware features mentioned above. Paging has also been implemented in the 645; although of immense help to the implementation, we do not regard paging as a concept fundamental to a description of the principles of the ideal memory simulation and shall postpone the discussion of paging until the end of this section.

Let us first examine how much of the ideal memory capability has been integrated into the hardware. Then a discussion of the software functions needed to compensate for those capabilities which are not provided by the hardware will follow.

4.1. Hardware Segmentation in the 645

Concepts of segment name, segment descriptor, and segment body have been integrated into the hardware as follows.

4.1.1. Segment Names. A segment name for the hardware is an integer s, called segment number, such that $0 < s < 2^{18}$.

4.1.2. Segment Descriptors. The segment descriptor of segment "s" is the s^{th} entry of a table called a Descriptor Segment. The descriptor segment is in core memory and its absolute address is kept in a processor register. A descriptor segment entry is called a Segment Descriptor Word (SDW). SDW number s will be designated by the notation SDW(s).

Attributes that can be recorded in an SDW area are:

- The absolute core address of the head of the segment body.

- The length of the segment body.

- Access rights for only one user with respect to the segment body.

- An invalid attribute flag. F, which, when ON, signals the absence of the above attributes in the SDW and causes the processor to fault.

Since an SDW can contain access rights for one user only, each user program must be provided with a private descriptor segment. (See Figure 2.)

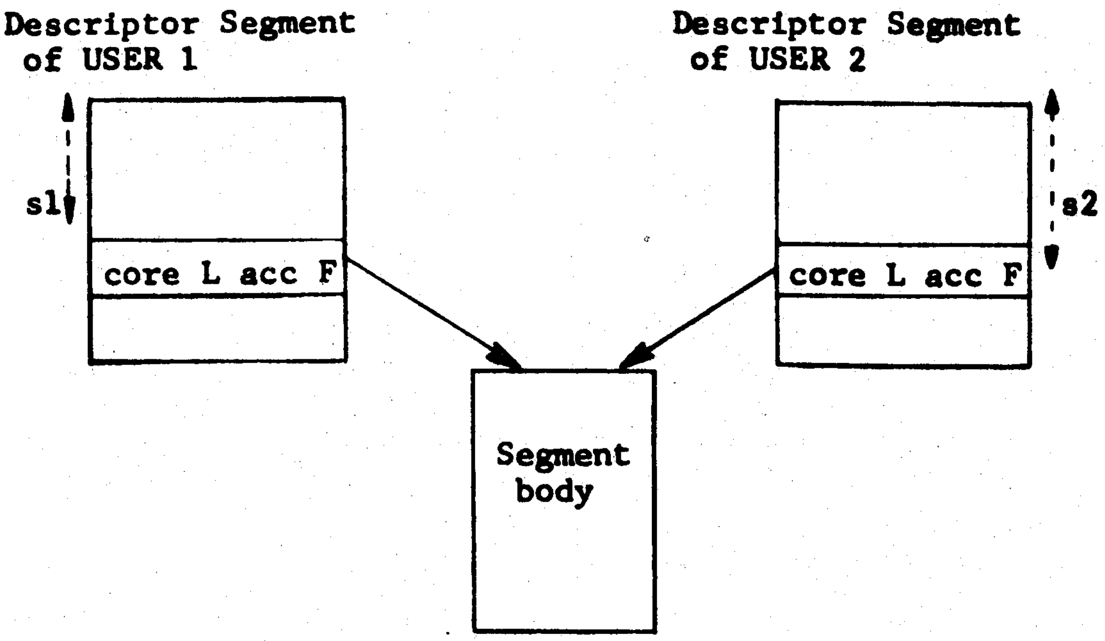


Figure 2. Hardware Segment Descriptors

4.1.3. Segment Bodies. The segment body is an array of contiguous words in core memory and its maximum length is 2^{18} words.

4.1.4. Address Transformations. Word number i of the body of segment s is addressed by the pair [s,i] and is accessed through SDW(s) by the processor.

Provided that the absolute core accress m_o of word 0 of the segment is stored in SDW(s), the processor transforms -

- the processor segment name s into the core memory address m_o using the descriptor segment which provides the mapping $m_o = Z(s)$.
- the processor address [s,i] into the core memory address m_i by the translation $m_i = m_o+i$, that is $m_i = Z(s)+i$.

4.1.5. Access Rights Checking. Before accessing word m_i the processor performs a check on -

- the length of the segment by comparing i to the length recorded in SDW(s).

- The access rights for the user with respect to segment s by using the access rights recorded in SDW(s).

This hardware organization presents the following advantages over more conventional hardware.

- The set of processor addresses [s,i] is sufficiently large that all words referenced by a program can be assigned unique processor addresses. The user does not have to organize a large program into overlays provided that he uses no more than 2^{18} segments.

- Processor addresses are independent of physical memory addresses. Addresses which appear in the instructions of a program are invariant when segments are moved from one location to another in core memory.

- Each access to core memory is subject to access rights checking.

However, the hardware has only a restricted understanding of the concept of segments and needs to be complemented by appropriate software features.

4.2. Software Segmentation

Given the foregoing hardware segmentation capabilities, the corresponding software segmentation capabilities required to implement the Multics virtual memory can be described.

4.2.1. Segment Names. A segment name is a character string called a symbolic segment name. The set of symbolic segment names is larger than 2^{18}. Therefore, the supervisor must map a large set of symbolic segment names into a smaller set of segment numbers.

4.2.2. Segment Descriptors. The hardware does not permit one to -

- retrieve attributes of a segment given the symbolic name of the segment. The software provides this capability.

- store all attributes of a segment in a hardware segment descriptor or SDW. The software provides complete segment descriptors for each segment and stores them in a catalog. See Figure 3.

Segment name	Segment Attributes			
a				
b				
c	core/secondary address	length	Access for user 1 Access for user 2 Access for user 3	other attributes
d				

Figure 3. Representation of a Catalog

4.2.3. Segment Bodies. The body of a segment is an array of contiguous words in core or in secondary memory. Since the processor can fetch data and instructions only from core-resident segments, the software must intercede when a segment is found to be missing from core.

4.2.4. Address Transformations. Assuming for the moment that all segments are in core memory, the supervisor performs the following three transformations to make segments accessible by the processor.

First, for any segment in the system, the supervisor must provide a one to-one mapping from its symbolic name n into its memory address m_o, where m_o is the address of the beginning of the segment. This mapping $m_o = X(n)^o$ is recorded in the catalog. See Figure 4.

Next, for each segment referenced by a user program, the supervisor must provide a one-to-one mapping from its symbolic name n into the segment number s assigned to it in this user program. This mapping $s = Y(n)$ is recorded in a table associated with the user program and called the Known Segment Table (KST).

Finally, for each segment that has been assigned a segment number s in a user program, the supervisor must provide a one-to-one mapping between the segment number s and its memory address m_o. This mapping $m_o=Z(s)$ is recorded in the descriptor segment associated with the user program.

The transformation X is independent of the user program; transformations Y and Z for user program u are user-dependent and will be denoted as Y_u and Z_u.

In order to permit several user programs to share the same segment by merely referencing it by the same name, these transformations must be such that, for any user program, $X(n) = Z_u(Y_u(n))$.

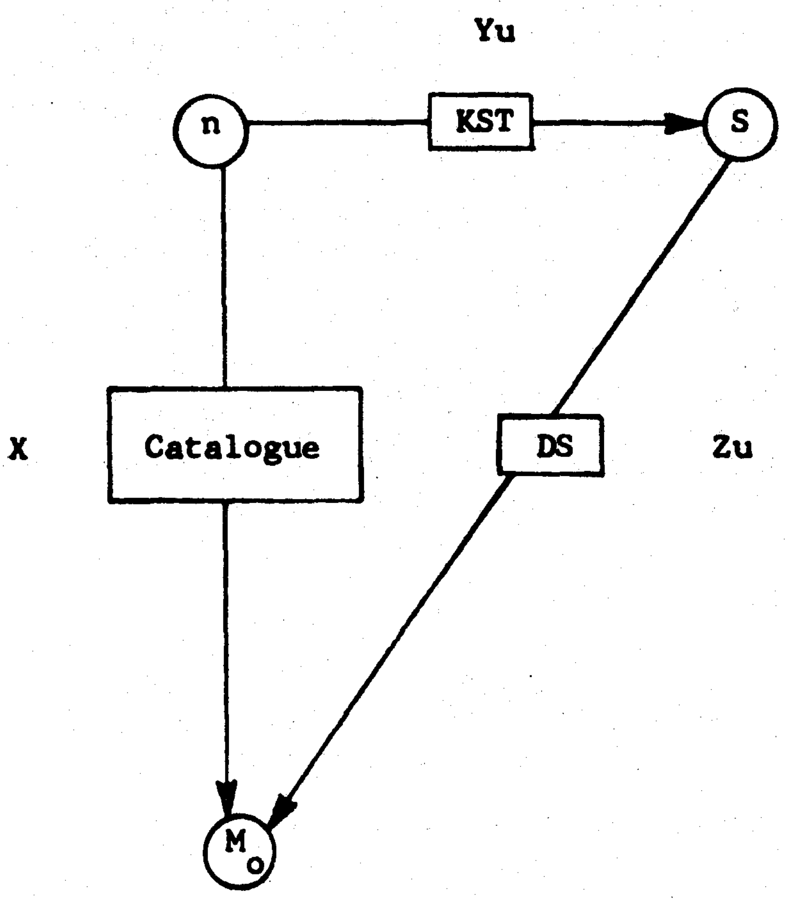


Figure 4. Address Mapping Tables

To this point we have assumed that all segments are in core. In fact, core memory being limited, the supervisor has to move segments between core and secondary memory.

The transportation of segment n from core memory address m_o to secondary memory address M_o must be associated with the following address mapping modifications:

- m_o must be replaced by M_o in the catalog entry for n.
- m_o must be replaced by an undefined value in any SDW in which it appears. This is done by setting the invalid attribute flag ON in the SDW.

Note that the mapping between n and s remains unchanged in any user program.

A subsequent reference to segment n by segment number s in a user program will cause the processor to fault since the invalid attribute flag is ON in SDW(s). This fault will be referred to as a missing segment fault. Using the KST associated with this user program, it is possible to determine the name n of the segment s. Knowing n, the catalog entry for n can be found. The segment must be moved from secondary memory address M_o to some (generally different) core memory address m_o'. This move must be associated with the following address mapping modifications:

- M_o must be replaced by m_o' in the catalog entry for n.
- The undefined value (Flag) in SDW(s) must be replaced by m_o'.

Note again that the mapping between n and s remains unchanged by the move.

4.2.5. Access Rights Checking. We have seen how the supervisor responds to a missing segment fault occurring in a user program but the description was not complete. A missing segment fault is a signal to evaluate the segment attributes in a specific SDW. Only the evaluation of the core address attribute has been described. Moreover, when the supervisor extracts core address information from the catalog, it also extracts the length and access rights attributes and stores them in the SDW. Each subsequent hardware reference to the segment by this user program is made through the SDW with the hardware performing access checking.

However, when performing operations on segment attributes the supervisor itself must do the necessary validation for any operation requested by a particular user since the hardware does not provide for access checking on attributes.

4.3. Paging

In a system in which the maximum size of any segment were very small compared to the size of the entire core memory, the "swapping" of complete segments into and out of core would be feasible. Even in such a system, if all segments did not have the same maximum size, or had the same maximum size but were allowed to grow from initially smaller sizes, there remains the difficult core management problem of providing space for segments of different sizes.

Multics, however, provides for segments of sufficient maximum size that only a few can be entirely core-resident at any one time. Also, these segments can grow from any initial size smaller than the maximum permissible size.

By breaking segments into equal-sized parts called pages and providing for the transportation of individual pages to and from core as demand dictates, several practical problems encountered in the implementation of a segmented virtual memory are solved.

First, since only the referenced page of a segment need be in core at one instant, segments need not be small compared to core memory.

Second, "demand paging" permits advantage to be taken of any locality of references peculiar to a program by transporting to core only those pages of segments which are currently needed. Any additional overhead associated with demand paging should of course be weighed against the alternative inefficiencies associated with dedicating core to entire segments which have been swapped into core but which may be only partly referenced.

Finally, since pages are all of equal size, space allocation is immensely simplified. The "compaction" of information in core and on secondary storage characteristic of systems dealing with variable-sized segments or pages is thereby eliminated.

The basic principles of paging in the Multics virtual memory may be briefly summarized as follows.

When a segment is not paged, the memory location of its element i is defined by relation (1), where m_o is the memory location of element 0.

$$(1) \quad m_i = m_o + i$$

When a segment is paged into pages of K elements, the memory location of its element i is defined by relation (2), where m_{pK} is the memory location of element pK; that is, the memory location of the page number p of the segment.

$$(2) \quad \begin{cases} m_i = m_{pK} + j \\ j = i \bmod K \\ p = (i-j)/K \end{cases}$$

If N is the number of pages in a segment, paging this segment requires -

- a <u>segment map</u> with N entries, one for each page.
- a relocation capability in the hardware.

In the 645 the N entries of the segment map are provided by a "page table" and the relocation is performed by the processor itself. Furthermore, a page table entry contains a <u>missing-page flag</u> such that, if found ON by the processor while attempting to perform relocation, causes the processor to trap to the supervisor.

The missing-page flag is ON when the corresponding page is not in core. When, upon attempting to access a missing page, the processor traps to the supervisor, the supervisor must move the requested page into core. In order to do so the supervisor must maintain a segment map of N entries in the software descriptor, i.e., in the directory entry. Each time page p is moved from one location to another, this move must be associated with the following address mapping modifications.

- Update the mapping in entry p of the segment map located in the directory entry.

- Update the mapping in entry p of the page table.

Although paging need not be considered essential to a description of the simulation principles of an ideal memory, it is a basic feature for the implementation of such a memory.

The next chapter describes in some detail how the ideal memory has been simulated in the Multics system, using hardware segmentation and hardware paging as implemented on the GE-645.

Chapter 2

IMPLEMENTATION OF THE MULTICS VIRTUAL MEMORY: OVERVIEW

1. INTRODUCTION

As we have seen in Chapter 1, the Multics virtual memory is a large, segmented memory. Each segment can be referenced by its name in a user program; a reference by name will cause the segment to be accessed by the processor according to the access rights of the user with respect to that segment. The memory is called "virtual" because it is not available as a hardware device. Instead, it is simulated using a conventional non-segmented memory, a set of processor registers which provide the second dimension of a segmented memory and a supervisor which compensates for the difference in capabilities between the 645 hardware and the ideal memory described in Chapter 1.

Although the hardware checks each user's access rights to a segment whenever it accesses that segment, a certain number of additional functions must be provided by the supervisor in order to give the illusion that all segments are directly accessible by name by the processor.

- The hardware cannot retrieve the attributes of a segment using its symbolic name; the supervisor organizes segment attributes into "directories" where it can retrieve them.

- The hardware cannot interpret access rights for segment attributes; all operations on segment attributes are done by the supervisor.

- The hardware cannot reference a segment by a symbolic name; it does it by a segment number. The supervisor translates all symbolic segment names into segment numbers.

- The hardware cannot access a segment if it is not in core memory; each reference to a segment which is not in core will cause the supervisor to move the segment from secondary memory to core memory. In order to help the supervisor in core memory allocation, the hardware provides a paging capability.

This chapter builds upon the ideas developed in Chapter 1 to show in some detail how the ideal memory is simulated. The major topics covered are:

- Segmentation and paging on the 645 processor.

- The organization of segment attributes into hierarchically ordered directories and the manipulation of these attributes by the supervisor.

- Segment accessing and all the supervisory functions needed to make a segment directly accessible by the processor.

- The structure of the supervisor itself, showing how parts of the supervisor are able to utilize the virtual memory provided for user programs.

2. THE 645 PROCESSOR

This paper discusses only those features of the 645 processor which are of interest for the implementation of a virtual memory. They can be grouped into two different classes -- segmentation and paging -- and are treated separately below.

2.1. Segmentation

Any address in the 645 consists of a pair of integers [s,i]. The range of s and i is 0 to $2^{18}-1$. s is called the segment number, i the index within the segment. Word [s,i] is accessed through a hardware register which is the s^{th} word in a table called a descriptor segment (DS). This descriptor segment is in core memory and its absolute address is recorded in a hardware register called a descriptor base register (DBR). Each word of the DS is called a segment descriptor word (SDW); the s^{th} SDW will be referred to as SDW(s). See Figure 1.

The DBR contains the following values:

- DBR.core which is the absolute core address of the DS.
- DBR.L which is the length of the DS.

Segment descriptor word number s contains the following values:

- SDW(s).core which is the absolute address of the segment s.
- SDW(s).L which is the length of the segment s.
- SDW(s).acc which describes the access rights for the segment.
- SDW(s).F which is a flag that can be ON or OFF. This is the invalid attribute flag mentioned in Chapter 1.

The algorithm used by the hardware for executing an instruction of the type OPCODE [s,i] is as follows:

- If DBR.L <s, generate a fault.
- Access SDW(s) at absolute location DBR.core + s.
- If SDW(s).F = ON, generate a missing segment fault.
- If SDW(s).L< i, generate a fault.
- If SDW(s).acc is incompatible with OPCODE, generate a fault.
- Apply OPCODE to the word whose absolute address is SDW(s).core+i.

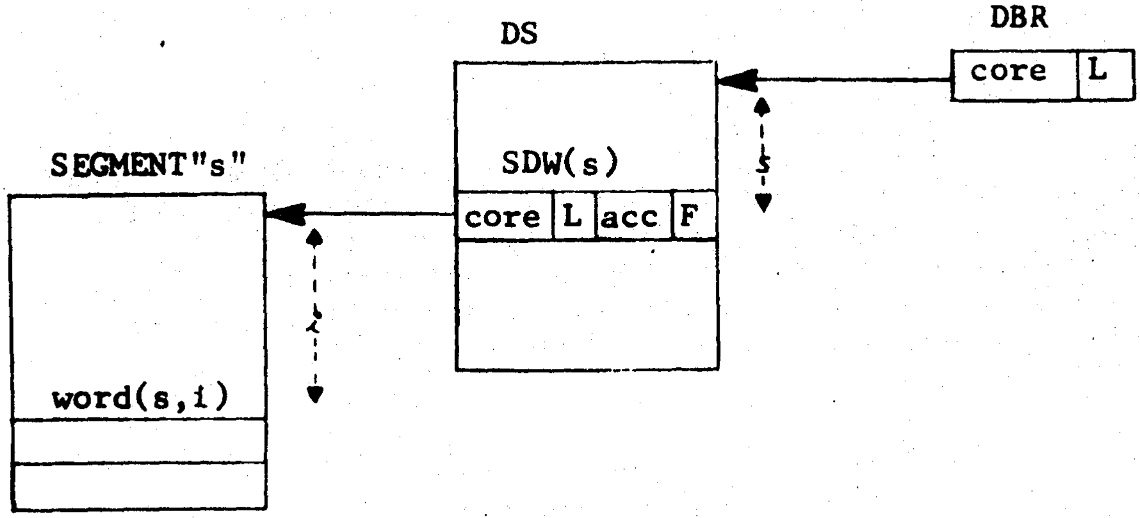


Figure 1. Hardware Segmentation in the 645

The above description assumes that segments are not paged; in fact, paging is implemented in the 645 hardware.

2.2 Paging

A bit in an SDW indicates whether the corresponding segment is paged or not. Another bit in the SDW indicates whether the page size is 64 or 1024 words. Analogous bits in the DBR serve the same purpose for the descriptor segment.

However, in the Multics implementation, all segments are paged and the page size is always 1024 words. Therefore, this description makes the following two assumptions:

- All segments are paged.
- The page size is a constant, K, equal to 1024 words.

No further reference will be made to these two bits in the SDW and DBR.

Element i of a segment is the yth word of the xth page of the segment, x and y being defined by:

$$\begin{cases} y = i \bmod K \\ x = (i-y)/K \end{cases}$$

where K is the page size.

Since K = 1024 = 2^{10}, the processor can compute x and y from the 18 bit-binary representation of i by merely dividing i into two parts. The right part, which consists of the 10 least significant bits of i, represents the binary value of y; the left part, which consists of the 8 most significant bits of i, represents the binary value of x.
See Figure 2.

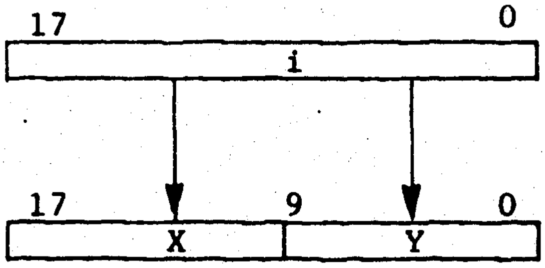


Figure 2. Hardware Interpretation of the Word Number

The page table (PT) of a segment is an array of physically contiguous words in core memory. Each element of this array is called a page table word (PTW).

Page table word number x contains the following items.

- PTW(x).core which is the absolute core address of page #x.
- PTW(x).F which is a flag that can be ON or OFF. This is the missing page flag mentioned in Chapter 1.

The meaning of DBR.core and SDW(s).core is now as follows:

- DBR.core = Absolute address of the PT of the DS.
- SDW(s).core = Absolute address of the PT of segment #s.

The full algorithm used by the hardware to access word [s,i] is (see Figure 3):

- If DBR.L < s, generate a fault.
- Split s into sx and sy such that sy = s mod K and sx = (s-sy)/K.
- Access PTW(sx) at absolute location DBR.core + sx.
- If PTW(sx).F = ON, generate a missing page fault.
- Access SDW(s) at absolute location PTW(sx).core + sy.
- If SDW(s).F = ON, generate a missing segment fault.
- If SDW(s).L < i, generate a fault.
- If SDW(s).acc is incompatible with OPCODE, generate a fault.
- Split i into ix and iy such that iy = i mod K and ix = (i-iy)/K.
- Access PTW(ix) at absolute location SDW(s).core + ix.
- If PTW(ix).F = ON, generate a missing page fault.
- Apply the OPCODE to the word whose absolute location is PTW(ix).core + iy.

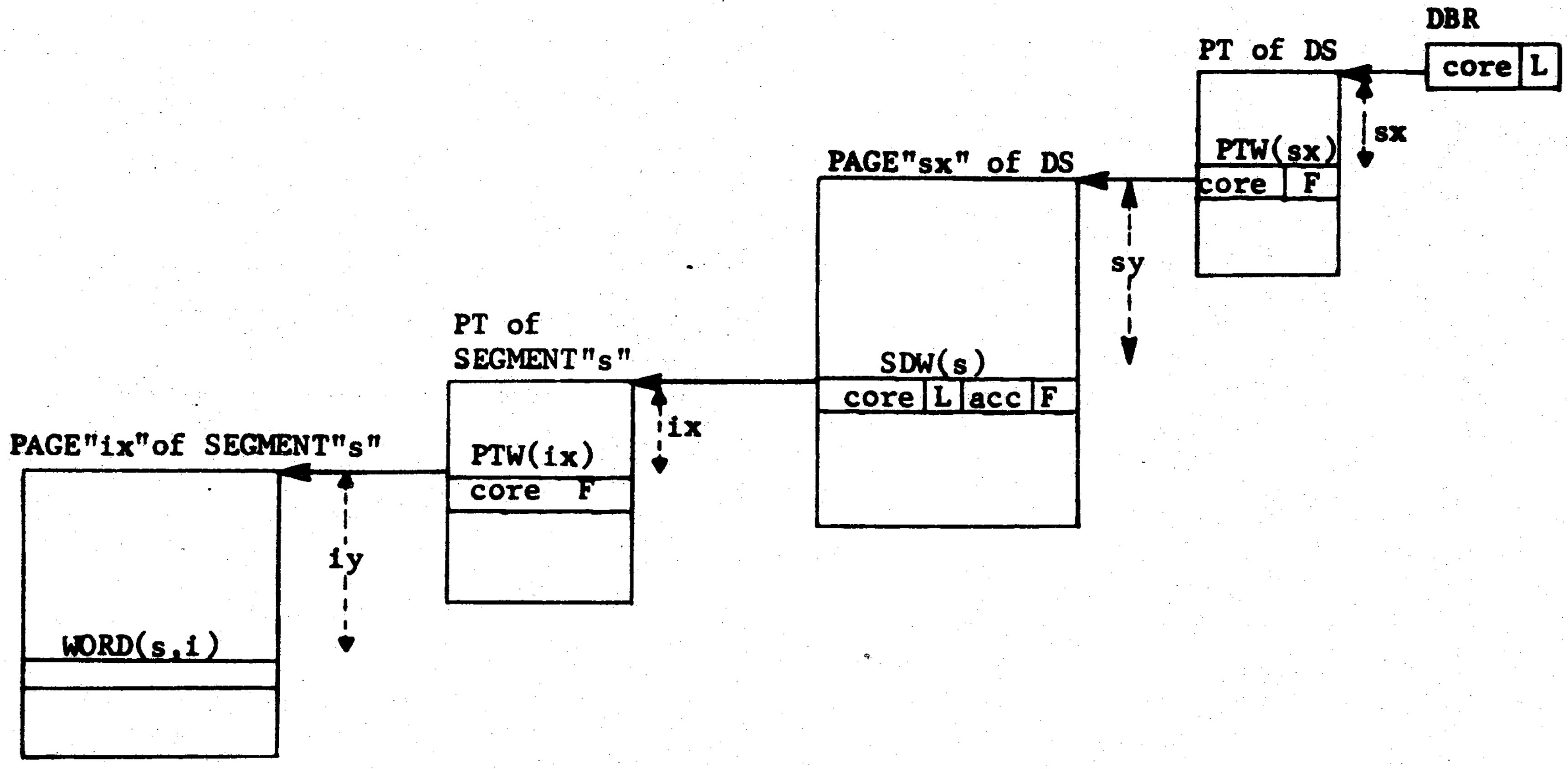


Figure 3. Hardware Segmentation and Paging in the 645

3. SEGMENT ATTRIBUTES

3.1. Directory Hierarchy

The association between the name of a segment and its attributes is recorded in a catalogue. This catalogue consists of a table with one entry for each segment in the system. An entry contains the name of the segment and all its attributes (length, memory address, list of users allowed to use that segment with their respective access rights, date the segment was created, etc.).

In Multics this catalogue is divided into several segments called directories, which are organized into a tree structure. A naming convention permits one to search the tree structure for a given name without having to search all directories.

A segment name is a list of subnames reflecting the position of the entry in the tree structure, with respect to the beginning of the tree or root directory. By convention, subnames are separated by the character ">". Each subname is called.an entryname and the list of entrynames is called a pathname.

There are two types of directory entries called branches and links. A branch is a directory entry which contains all attributes of a segment while a link is a directory entry which contains the pathname of another directory entry. This chapter will deal only with entries of the branch type.

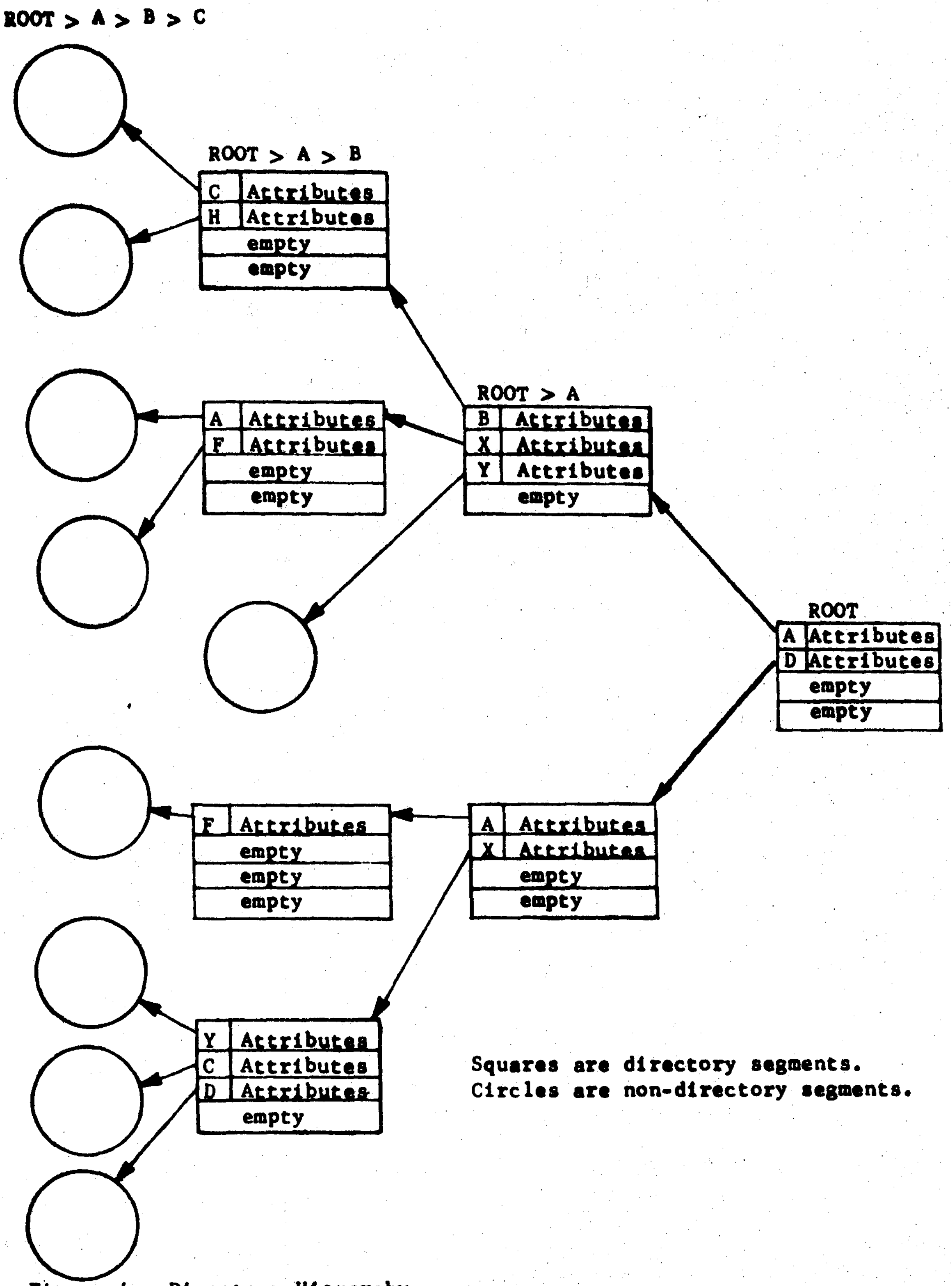
The pathname is the only name by which a segment can be searched for in the directory hierarchy.

The attributes associated with a segment whose pathname is ROOT > A > B > C are found as follows (see Figure 4):

- Search the root directory for an entry whose entry name is A. This entry contains attributes for the directory segment whose pathname is ROOT > A. These attributes permit one to locate the directory ROOT> A in memory.

- Search directory ROOT > A for an entry whose entry name is B. This entry contains attributes for the directory segment whose pathname is ROOT > A > B. These attributes permit one to locate the directory ROOT > A > B in memory.

- Search directory ROOT > A > B for an entry whose entry name is C. This entry contains attributes for the segment ROOT > A > B > C.



Squares are directory segments.
Circles are non-directory segments.

Figure 4. Directory Hierarchy

3.2. Operations on Segment Attributes

All operations on segment attributes are done by supervisor primitives. There is a set of primitives available to the user which allow him, for example, to:

- Create a segment.
- Delete a segment.
- Change the entryname of a directory entry.
- Change the access rights of a segment.
- List a directory.

Any of these operations is performed on behalf of a user by the supervisor only if the user has the right to perform them.

Some further details about one of these operations, segment creation, are important to an understanding of the topic of segment accessing developed in the next section.

Creating a segment whose pathname is ROOT > A > B > C consists basically of taking the following actions:

- Check, by searching the directory hierarchy, that this segment does not exist already in the system.

- Allocate space for a branch in directory ROOT > A > B.

- Store in the branch the following items:

  - The entry name C.

  - The access list, given by the creator.

  - The segment map which consists of a secondary storage address for each page of the segment. This segment map is manufactured by the supervisor.

  - The segment status "inactive", meaning that there is no page table for this segment.

Once the segment has been created, the user can reference it. Note that no segment number has been assigned to the segment at creation time, and the only way to refer to it is by the pathname.

4. SEGMENT ACCESSING

We are now in a position to understand a description of the functions that are provided by the supervisor in order to make accessible by the processor segments which are referenced by name in a user program. Figure 5 is key to an appreciation of the Multics virtual memory implementation. Although frequent references to Figure 5 follow, the full implications of its contents will not be apparent until the entire section has been read.

4.1. Concept of Process and Address Space

A process is generally understood as being a program in execution. A process is characterized by its state-word defining, at any given instant, the history resulting from the execution of the program.

A process is also characterized by its address space. The address space is the set of processor addresses that this process can use to reference the memory. In Multics the address space of a process is defined as the set of segment numbers that the process can use to reference segments in the virtual memory. As explained in Chapter 1, a segment number can be used to reference the virtual memory only if it has been associated with a segment name, i.e., a pathname. This association [pathname, segment number] is recorded in a table called the Known Segment Table (KST) which defines the address space of the process. There is a one-to-one correspondence between Multics processes and address spaces. The action of adding a new pair [pathname, segment number] in a KST is referred to as making the segment with that pathname known to the process.

4.2. Making a Segment Known to a Process

Each time a segment is referenced in a process by its pathname, the pathname must be translated into a segment number in order to permit the processor to address the segment. This translation is done by the supervisor using the KST associated with the process. The KST is an array organized such that the entry number "s", KSTE(s), contains the pathname associated with segment number s. See Figure 5.

If the association [pathname, segment number] is found in the KST for this process, then the segment is known to the process and the segment number can be used to reference the segment.

If the association [pathname, segment number] is not found it means that this is the first time the segment is referenced in the process and the segment has to be made known. This is done by assigning an unused segment number "s" in the process and by establishing the pair [pathname, segment number] in the KST by recording the pathname in KSTE(s). Furthermore, the directory hierarchy is searched for this pathname and a pointer to the corresponding branch is entered in KSTE(s) for later use (see Section 4.3.).

This stage is fundamental because, in the Multics system, it is impossible to assign a unique segment number to each segment. The reason is that the number of segments in the system may be larger than the number of segment numbers available in the processor.

When a segment is made known to a process by segment number "s" its attributes are not placed in SDW(s) of the descriptor segment of that process. SDW(s) has been initialized with an invalid attribute flag. Therefore, the first reference in this process to that segment by segment number "s" will cause the processor to generate a fault. In Multics this fault is called a "missing segment fault" and transfers control to a supervisor module called the segment fault handler.

4.3. The Segment Fault Handler

Upon the occurrence of a missing segment fault, control is passed to the segment fault handler whose function is to store the proper segment attributes in the appropriate SDW and to set the invalid attribute flag OFF in the SDW.

This information, we recall, consists of:

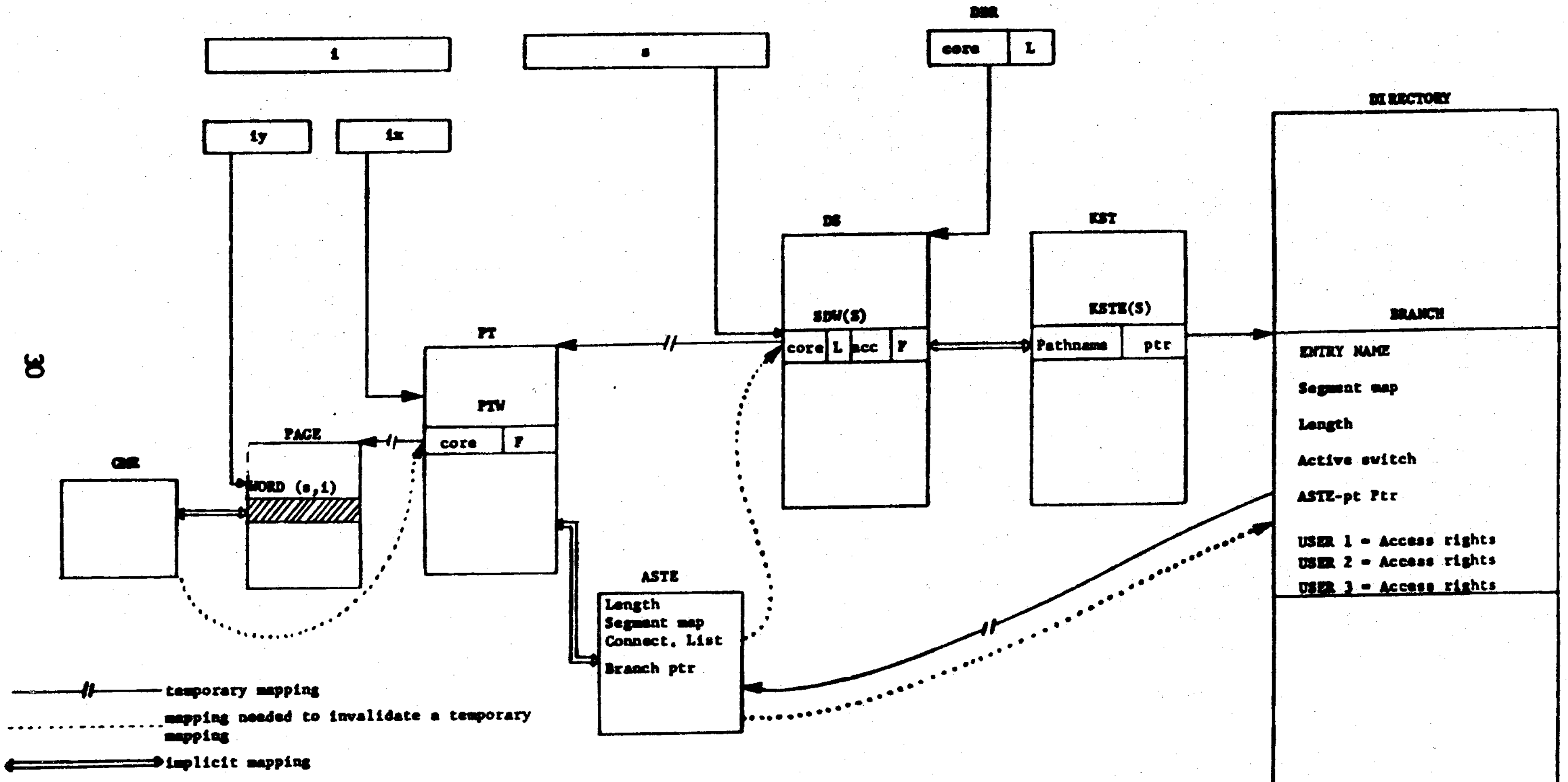
- The page table address.
- The length of the segment.
- The access rights of the user with respect to the segment.

The information initially available to the supervisor upon occurrence of a missing segment fault is:

- The segment number s.
- The process identification.

The only place where the needed information can be found is in the branch of the segment. Using the process identification, the supervisor can find the KST for this process. It can then search this KST for the segment number s. Having found the KST entry for s, it can find the required branch since a pointer to the branch has been stored in the KST entry when the segment was made known to that process. See Section 4.2.

Using the active switch (see Figure 5) in the branch, the supervisor can determine whether or not there is a page table for this segment. Recall that this switch was initialized in the branch at segment creation time. If there is no page table, one must be constructed. A portion of core memory is permanently reserved for page tables. All page tables are of the same length and the number of them is determined at system initialization.



NOTE - The page table of the descriptor segment is not shown for the sake of simplicity

Figure 5. Basic Tables Used to Implement the Multics Virtual Memory

The supervisor divides these page tables into two lists: the "used list" and the "free list". Manufacturing a page table (PT) for a segment could consist only of selecting a PT from the free list, putting its absolute address in the branch and moving it from the free to the used list. If this were actually done, however, then the servicing of each missing page fault would require access to a branch since the segment map is kept there.

Since all directories cannot be core-resident, page fault handling could thereby require a secondary storage access in addition to the read required to transport the page itself into core. Although this mechanism works, efficiency considerations have led to the "activation" convention between the segment fault handler and the page fault handler.

4.3.1. Activation. A portion of core memory is permanently reserved for recording attributes needed by the page fault handler, i.e., the segment map and the segment length. This portion of core is referred to as the active segment table (AST). The AST contains one entry (ASTE) for any segment that has a PT. A PT is always associated with an ASTE, the address of one implying the address of the other. They may be regarded as a single entity and will be referred to as the [PT,ASTE] of a segment.

A segment which has a [PT,ASTE] is said to be active. The property of being active or not active is an attribute of the segment and, therefore, has to be recorded in the branch. When this active switch is set ON it means that both the segment map and the segment length are no longer in the branch but are to be found in the segment's [PT,ASTE] whose address has been recorded in the branch during "activation" of the segment.

To activate a segment the supervisor must:

- Find a free [PT,ASTE]. Assume temporarily that at least one is available.

- Move the segment map and the segment length from the branch into the ASTE.

- Set the active switch in the branch.

- Record the pointer to [PT,ASTE] in the branch.

Having defined activation, the actions taken up to now by the segment fault handler can be summarized as:

- Use the segment number s to access the KST entry.

- Use the KST entry to find the branch.

- If the active switch is OFF, activate the segment. If it is ON, then activation is unnecessary at this time as the segment was already activated for another process.

By pairing an ASTE with a PT in core, the segment fault handler has guaranteed that the segment attributes needed by the page fault handler are core-resident, thus permitting efficient page fault servicing.

4.3.2. <u>Connection</u>. Now that the segment is active, the corresponding SDW must be "connected" to the segment.

To connect the SDW to the segment the supervisor must:

- Get the absolute address of the PT, using the [PT,ASTE] pointer kept in the branch, and store it in the SDW.

- Get the segment length from the ASTE and store it in the SDW.

- Get the access rights for the user from the branch and store them in the SDW.

- Turn off the flag which caused the fault from the SDW.

Having defined activation and connection, segment fault handling can finally be summarized as:

- Use the segment number s to access the KST entry.
- Use the KST entry to find the branch.
- If the active switch in the branch is OFF, activate the segment.
- Connect the SDW.

Note that segment sharing in core is "automatically" guaranteed by the use of the active switch and [PT,ASTE] pointer kept in the segment branch since all SDW's describing this segment will point to the same PT.

Now that the segment has an SDW pointing to the PT, the hardware can access the appropriate page table word. If the page is not in core, a missing page fault occurs, transferring control to the supervisor module called the page fault handler.

4.4. The Page Fault Handler

When a page fault occurs the page fault handler is given control with the following information:

- The PT address.
- The page number.

The information needed to bring the page into memory is:

- The address of a free block of core memory into which the page can be moved.
- The address of the page in secondary memory.

A free block of core must be found. This is done by using a data base called the core map. The core map is an array of elements called core map entries (CME). The nth entry contains information about the nth block of core (the size of all blocks is 1024 words). The supervisor divides this core map in two lists; the used list and the free list.

The job of the page fault handler is to:

- Find a free block of core. (Assume temporarily that there is at least one free block in the free list.)
- Access the ASTE associated with the PT and find the address in secondary memory of the missing page.
- Issue an I/O request to move the page from secondary memory into the free block of core.
- Upon completion of the I/O request, store the core address in the PTW and remove the fault from the PTW.

4.5. Page Multiplexing

It was assumed that a free block of core was available in the core map free list; however, this is not always the case since there are many more pages in the virtual memory than there are blocks of core. Therefore, in order to get a free block of core, the page fault handler may have to move a page from core to secondary memory. This requires:

- An algorithm to select a page to be removed.
- Knowing the address of the PTW which holds the address of the selected page in order to set a fault in it.
- Knowing where to put the page in secondary memory.

The selection algorithm is based upon page usage. The hardware provides valuable assistance by the fact that, each time a page is accessed, a bit is set ON in the corresponding PTW. This bit is called the <u>used bit</u>.

The selection algorithm will not be described here; however, it should be noted that candidates for removal are those pages described in the core map used list. Therefore, each core map entry which appears in the used list must contain a pointer to the associated PTW in order to permit one to examine the used bit. The action of storing the PTW pointer in the core map entry must be added to the list of actions taken by the page fault handler when a page is moved into core (see Section 4.4.).

- A fault is stored in the PTW.

- The secondary storage address for the page is found in the ASTE whose address can be computed from the PTW address.

- An I/O request is issued to remove the page to secondary storage.

- Upon completion of the I/O request, the core map entry is removed from the used list and put in the free list.

By this mechanism, blocks of core are multiplexed among all pages of all active segments in the system.

It is important to realize that a page is either in core or in secondary storage. There is no such thing as a "copy" of a page. When a page is moved from secondary storage to core, its secondary storage address, located in the ASTE, could be freed; it is no longer needed since the address of the page is now in the PTW. When the page is to be removed, a free block of secondary storage could be assigned to it. It is only for practical reasons that the block of secondary storage is not freed each time a page is moved into core.

Page multiplexing maintains a "perpetual motion" between core and secondary storage of pages of active segments. If the set of active segments in the system were invariant, then pages of other segments would never have a chance to be in core.

4.6. [PT,ASTE] Multiplexing

In the description of segment fault handling, when a segment had to be activated, a pair [PT,ASTE] was assumed available for assignment to that segment. In fact, the number of [PT,ASTE] pairs is limited in the system and is, by far, smaller than the number of segments in the virtual memory. Therefore, these [PT,ASTE] pairs must be multiplexed among all segments in the virtual memory.

This means that making a segment active may imply making another segment inactive thereby disassociating this other segment from its [PT,ASTE]. Since each process sharing the same segment will have the address of the PT in an SDW it is essential to invalidate this address in all SDW's before removing the page table. It is also essential to move to secondary memory all pages of that segment which are in core before removing the [PT,ASTE], since the ASTE is needed to remove a page. Then, and only then, can [PT,ASTE] be disassociated from the segment.

This operation requires:

- An algorithm to select a segment to be deactivated.
- Knowing all SDW's that contain the address of the page table of the selected segment in order to invalidate this address.
- The removal of all pages of the selected segment that are still in core.
- Moving the attributes contained in the ASTE back to the branch and changing the status of the segment from active to inactive in the branch.

The selection algorithm is here again based on segment usage. The only thing of interest at this point is that selection is done by scanning the ASTE used list. Therefore, the ASTE must provide all the information needed for removing the [PT,ASTE]. This means that during activation and connection this information must be made available as explained below.

During activation, a pointer to the branch must be placed in the ASTE; during connection, a pointer to the SDW must be placed in the ASTE. Since more than one SDW is connected to the same PT when the segment is shared by several processes the supervisor must maintain a list of pointers to connected SDW's. This list is called a connection list. See Figure 5.

Now we are in a position to understand how a [PT,ASTE] can be disassociated from a segment. After the selection algorithm decides on an ASTE to be freed, actions to be taken consist of two steps called "disconnection" and "deactivation".

Disconnection consists of storing a segment fault in each SDW whose address appears in the connection list in the ASTE.

Deactivation consists of removing all pages of this segment that may be in core, moving the segment map from the ASTE back to the branch, resetting the active switch in the branch and putting the [PT,ASTE] in the free list.

4.7. Segment Number Multiplexing in a Process

The number of segments that a process can describe in its descriptor segment is limited to 2^{18}. It is unlikely that a process needs to access more than 2^{18} segments from the time it is created to the time it is destroyed. However, if this should happen, a facility is provided to a process to remove an association [pathname, segment-number] by an explicit call to the supervisor. This action is referred to as making a segment unknown. When segment A which is known to a process by the segment number "s" is made unknown to that process, no attempt is made by the supervisor to remove residual [s,i] pairs that may have been generated and stored during the time that s was assigned to A. Making segment A unknown to the process implies freeing KSTE(s). If subsequently another segment, say B, is made known to the process, the supervisor may assign this unused segment number s to segment B, entering the pathname B in KSTE(s). From this point on, any reference by segment number s in this process will cause segment B to be accessed. Therefore, it is entirely the responsibility of the programmer, after segment A is made unknown, not to reuse any residual pair [s,i] that was generated for accessing segment A.

4.8. Directory Entry Multiplexing

When a segment is deleted, the branch of that segment is deleted. No attempt is made by the supervisor to remove residual KST entries that contain a pointer to this branch. However, the supervisor can detect references by residual segment numbers, to segments which have been destroyed as follows:

- When segment A is created, the supervisor assigns a unique number N_A to segment A and stores it in the branch.

- When segment A is made known to a process P by the segment number s, N_A is copied from the branch to KSTE(s) for process P along with the pointer to branch A.

- If segment A is deleted by any process, the supervisor disconnects the corresponding SDW in process P, if it was connected, and deletes the branch together with N_A.

- If the same directory entry is reused to record the branch information of another segment B, a new unique identifier N_B will be stored in the branch.

- Now, if process P uses the segment number s in order to access segment A, a segment fault will occur; the KST entry contains a pointer to the directory entry which is supposed to be the branch A. But by comparing the N_A of the KST entry and N_B of the directory entry, the supervisor can discover that segment A has been deleted.

Therefore, it is possible to detect the deletion of a branch even though its former directory entry has been reused for another segment.

5. STRUCTURE OF THE SUPERVISOR

Up to now supervisor functions have been described, but supervisor structure has not been discussed. In this section, the different components of the supervisor are covered and the ability of portions of the supervisor to partially utilize the virtual memory is demonstrated.

5.1. Functional Modules

Three functional modules can be identified in the supervisor described; they are called directory control (DC), segment control (SC), and page control (PC).

5.1.1. Directory Control. Directory control is that part of the supervisor which can manipulate all segments in the system. DC identifies a segment by its pathname which uniquely defines a segment in the system. Data bases that are manipulated by DC are the directories and KST's of all processes (see Figure 6). DC provides all primitives to simulate operations on segment attributes; it also provides the assignment of a segment number to a segment within a process.

5.1.2. Segment Control. Segment control is that part of the supervisor which can manipulate only those segments which are known to at least one process. SC identifies a segment by either its segment number within a particular process, which uniquely defines a segment in the system, or by its [PT,ASTE] address which uniquely defines an active segment in the system. Data bases that are manipulated by SC are directories, KST's of all processes, descriptor segments of all processes and [PT,ASTE] pairs of all active segments. SC provides the functions of activation, connection, disconnection and deactivation.

5.1.3. Page Control. Page control is that part of the supervisor which can manipulate only those segments which are active. PC identifies a segment by its [PT,ASTE] address which uniquely defines an active segment in the system. Data bases that are manipulated by PC are [PT,ASTE] pairs of all active segments and the core map. PC provides the mechanism to move pages of active segments between secondary storage and core.

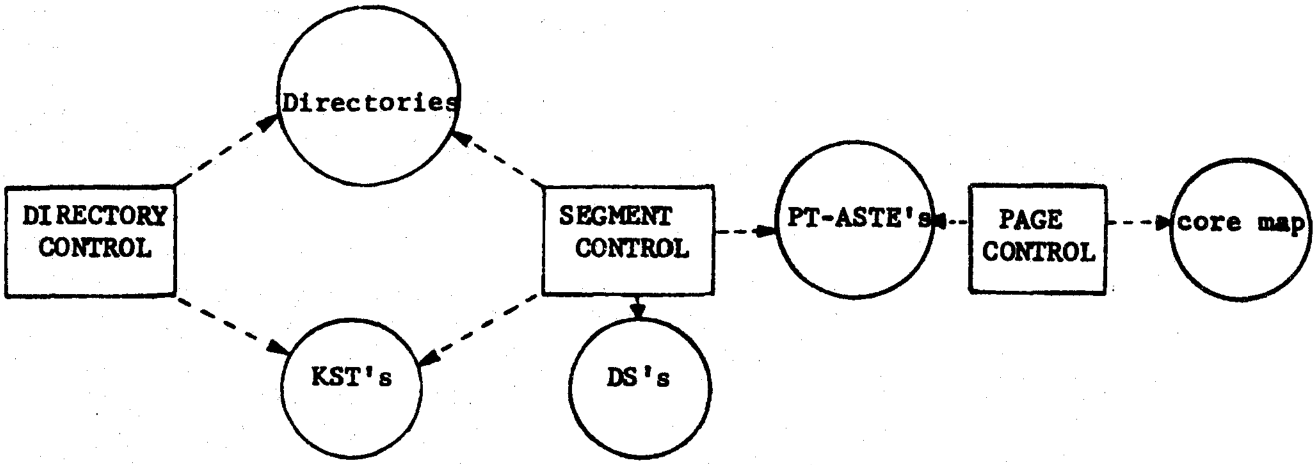


Figure 6. Supervisor Functional Modules and Data Bases

5.2. Use of Segmentation in the Supervisor

Previous to this no assumptions were made about the type of addressing used by the supervisor. It could be written so as not to use segment addressing of course; but organizing the supervisor into procedures and data segments permits one to use in the supervisor the same standard conventions that are used in a user program. For instance, the CALL-SAVE-RETURN conventions made for user programs can be used by the supervisor, the standard way to manufacture pure procedures in a user program can be used in the supervisor, etc. Thus, it seems desirable to use segmentation in the supervisor, and the following (temporary) assumption will be made:

Assumption 1:

a. The address space of the supervisor is entirely defined by a descriptor segment.

b. All segments used by the supervisor are always in core.

Assumption 1.b is not realistic, however, since it generally is not possible to dedicate enough core to contain the entire supervisor. It is, therefore, interesting to determine whether there is a way to use the page fault handler to transport supervisor as well as user pages.

5.3. Use of PC in the Supervisor

For the purpose of this paper, let us assume the validity of the following statement. "Page fault handling for a page x must be performed without referencing page x".

It is certainly possible to design a PC module which allows recursive page faults provided that the above condition is always satisfied. Each recursive invocation of the page fault handler should use a set of pages which does not include any of the pages that caused the previous invocations; furthermore, the number of recursive invocations must be guaranteed to be finite. The technique that has been chosen in Multics for page fault handling is to fix this finite number to 1 and thus no recursive page faults will ever occur. This decision has been made for reasons of efficiency and design simplicity. Therefore, it is assumed that all segments used in PC are always in core.

We can observe that the page fault handler need not know if a missing page belongs to a user or to the supervisor; it only expects to find the information it requires in the [PT,ASTE] of the segment to which the missing page belongs. Therefore, if all segments used in SC and DC are always active, then their pages need not be in core since PC can load them when they are referenced.

Thus, assumption 1 can be replaced by the following one (again temporary):

Assumption 2:

a. The address space of the supervisor is entirely defined by a descriptor segment.

b. All segments used in PC are always in core.

c. All segments used in SC and DC must be active and connected.

This convention turned out to be satisfactory in the Multics implementation except for directories. Recall that segments used by SC and DC are: (a) SC and DC procedures, (b) KST's and DS's, and (c) Directories.

The number of segments in class (a) and (b) is relatively small. On the contrary, the number of directory segments may be very large and keeping them always active is not a realistic approach, since a large number of [PT,ASTE] pairs would have to be permanently assigned to them. Therefore, it is desirable to use SC to activate and connect directory segments.

5.4. Use of Segment Control in the Supervisor

A necessary condition for handling a segment fault for segment x in a process is that segment x be known to that process. If SC is to handle segment faults taken by the supervisor for directories, all directories must be known to the supervisor. This means that the address space of the supervisor must be defined not only by its descriptor segment but also by KST, which contains one entry for each directory. After its KST has been so initialized, the supervisor looks like any other process.

Assuming that all directories are known to the supervisor process, but not necessarily active, a supervisor reference to a directory x may cause a segment fault. Recall that when handling this fault, the segment fault handler must reference the parent directory of segment x, where the branch for x is located. This reference to the parent of x could, in turn, cause a recursive invocation of the segment fault handler. Recursive invocations can propagate from directory to parent directory up to the root. If there is a way of stopping the recursion, then any segment fault on directories can be handled.

One way of stopping the recursion is to keep the root active and connected, so that a segment fault never occurs for it.

Assumption 2 can now be replaced by assumption 3, again temporary.

Assumption 3:

a. The address space of the supervisor process is defined by a descriptor segment and a KST.

b. All segments used in PC are always in core.

c. All segments used in SC and DC are always active and connected, except directories.

d. The root directory is always active and connected.

e. All directories are known to the supervisor process.

However, it is unsatisfactory to keep all directories known. We would like to keep known only those which may possibly be used in a segment fault handling, provided that other directories can be made known by directory control when needed.

5.5. <u>Use of the Make Known Facility in the Supervisor</u>

Making a segment x known implies searching for its pathname in the KST. If not found, the parent of x must first be made known and so on up to the root. If the root directory is always known to the supervisor, then any directory can be made known to the supervisor by the supervisor itself.

Assumption 3 will now be replaced by the final assumption:

<u>Final Assumption</u>:

a. The address space of the supervisor process is defined by a descriptor segment and a KST.

b. All segments used in PC are always in core.

c. All segments used in SC and DC, except directories, are always active and connected.

d. The root directory is always active and connected.

e. If a segment is known to any process, its parent directory must be known to the supervisor process.

f. The root directory is always known to the supervisor process.

Given the above assumptions, supervisor segments as well as user segments can be stored in the virtual memory that the supervisor provides.

5.6. The Supervisor Address Space

Unlike most supervisors, the Multics supervisor does not operate in a dedicated process or address space. Instead, the supervisor procedure and data segments are shared among all Multics processes. Whenever a new process is created, its descriptor segment is initialized with descriptors for all supervisor segments allowing the process to perform all of the basic supervisory functions for itself. The execution of the supervisor in the address space of each process facilitates communication between user procedures and supervisor procedures. For example, the user can call a supervisor procedure as if he were calling a normal user procedure. Also, the sharing of the Multics supervisor facilitates simultaneous execution, by several processes, of supervisory functions, just as the sharing of user procedures facilitates the simultaneous execution of functions written by users.

Since supervisor segments are in the address space of each process, they must be protected against unauthorized references by user programs. Multics provides the user with a ring protection mechanism which segregates the segments in his address space into several sets with different access privileges. The Multics supervisor takes advantage of the existence of this mechanism and uses it, rather than some other special mechanism, to protect itself.

6. SUMMARY

If only a few points discussed here were to be remembered, they should be those mentioned below. They have been separated into two classes: the point of view of the user of the virtual memory, and the point of view of the supervisor itself.

User Point of View

- The Multics virtual memory is capable of containing a very large number of segments that can be identified by their symbolic names.

- Segment attributes are stored in special segments called directories, which are organized into a tree structure; there is a naming convention, of which the user must be aware, by which a segment name must be the pathname of its branch in the directory tree structure.

- Any operation on directory segments must be done via a call to the supervisor.

- Any operation on a non-directory segment can be done directly in accordance with the access rights that the user has for this segment; any word of any segment which resides in the virtual memory can be referenced with a pair [pathname,i] by the user.

- A process can have only a limited number of segments in its address space. If the programmer wants to overlay a segment A by a segment B in the process address space, he can call the supervisor to do it but he must be aware of the dangers that this operation may present.

Supervisor Point of View

- The supervisor must simulate a large segmented memory directly addressable by segment name such that any access to the memory is submitted to access rights checking.

- It maintains a directory tree where it stores all segment attributes. It can retrieve the attributes of a segment given the pathname of that segment.

- The supervisor itself is organized into segments and runs in the user process address space.

- Any segment, be it a directory or a non-directory segment, is identified by its pathname but can be accessed only using a segment number. For each segment name the supervisor must assign a segment number by which the processor will address the segment in the process.

- The processor accesses a word of a segment through the appropriate SDW and PTW and subject to the access rights recorded in the SDW.

- A segment fault is generated by the processor whenever the page table address or access rights are missing in the SDW. The supervisor then, using the KST entry as a stepping stone, accesses the branch where it finds the needed information. If a PT is to be assigned, the supervisor may have to deactivate another segment.

- A page fault is generated by the processor whenever a PTW does not contain a core address. The supervisor then, using the ASTE associated with the PT, moves the missing page from secondary storage to core. This may require the removal of another page.

Chapter 3

DIRECTORY STRUCTURE

1. INTRODUCTION

A virtual memory system must include some means of storing and retrieving information. A segment is the unit of information in the Multics virtual memory which is so stored and retrieved.

All information about a segment such as its length and its location are called "attributes" of the segment. If the attributes of a segment can be located, then the segment itself can be found.

The attributes of segments are stored in special segments called "directories" and the directories are organized into a tree structure called "the directory hierarchy". All of the attributes of one segment are recorded in one entry in a directory. The entries in a directory can be referenced by literal string names called "entrynames".

The discussion which follows gives some of the details of the directory hierarchy structure, the naming of entries and segments and the contents of directory entries. Segment creation and deletion are also described since those operations are closely related to the creation and deletion of the attributes of a segment.

Other chapters describe the details of the search for a segment and how segments themselves are handled.

2. THE DIRECTORY HIERARCHY AND TERMINOLOGY

2.1. The Structure

A tree structured directory hierarchy is shown in Figure 1. Directory segments are shown as squares and non-directory segments are shown as circles. The lines between segments are branches of the tree structure and in Multics, denote the fact that the attributes of a segment at the lower end of a line are recorded in an entry in the directory at the upper end of the line. Thus the attributes of the segment labeled C in Figure 1, are recorded in the directory labeled B. The directory entries in which the attributes of segments are recorded are called "branch entries" or "branches".

A directory is said to be the parent of a segment if it contains the branch with the attributes of that segment. The parent directory of a segment is said to be "immediately superior to" the segment and the segment is "immediately inferior to" its parent. In Figure 1, the directory at the top of the structure labeled "root" and called "the root directory" or simply "the root" is the parent of or immediately superior to the segment labeled D. The root is superior to the segment labeled E, but not immediately superior to E. The segment labeled E is inferior to the root and the segment labeled D is immediately inferior to the root.

The root is the starting point in the search for segments. Note that the root has no branch. Its attributes, among them its location, are assumed to be known to the modules which perform the search.

There is one and only one branch per segment in the Multics system. This rule arose from the difficulty of finding and updating all the branches of a segment if one of its attributes should be changed. For cases in which it is useful to have a branch in more than one directory a link (see below) can be used.

2.2. Entrynames and Pathnames

An entryname is used to locate an entry in a given directory, but a "pathname" is needed to search the directory hierarchy for a particular entry. In order to uniquely locate a particular entry in a directory, an entryname must be unique in that directory. However, an entry can have several names (synonyms).

A pathname is the concatenation of an ordered sequence of entrynames. The entries must be located in the order they were named in order to follow the path from the root to the desired entry. The entrynames are separated by the character ">".

The name of a segment is the pathname which addresses its branch.

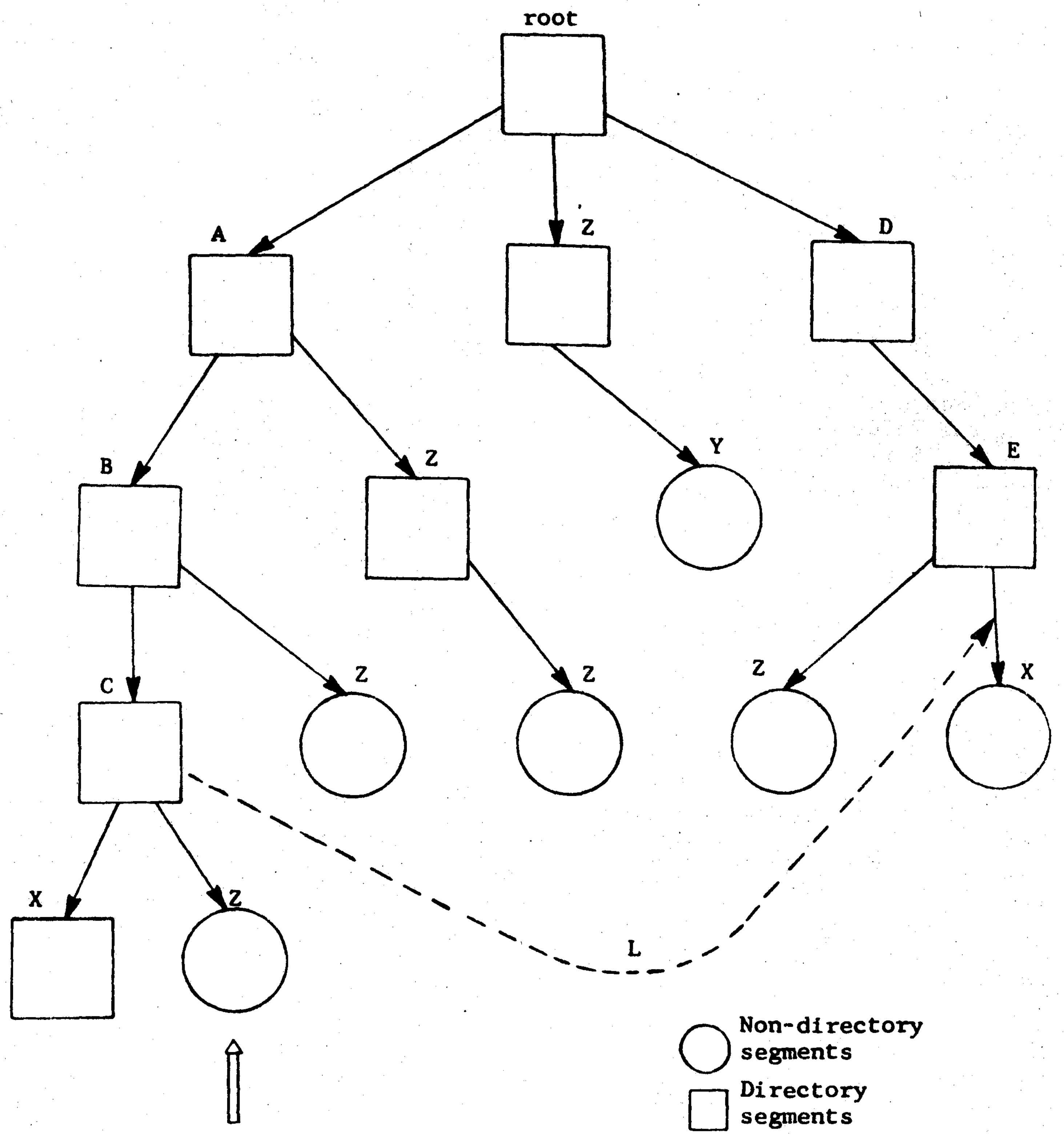


Figure 1. Tree structure directory hierarchy

The names root > A and root > D > E > Z are examples of pathnames for segments in the hierarchy shown in Figure 1. The name root > A > B > C > Z is the name of the segment pointed to by the vertical arrow. Since all pathnames begin with root >, the leading symbol > is used to mean root >. The names above become > A, > D > E > Z and > A > B > C > Z.

An example of the search for the attributes of the segment > B > Z in Figure 2 is instructive. The root is searched for a branch with the name B. The segment > B is accessed using the information in this branch. Next, the directory segment > B is searched for a branch with the name Z. When the branch Z is found in the directory > B then the search is finished.

It must be emphasized that the only way in which the search modules can find a segment is through use of a pathname.

A pathname can have synonyms since the entrynames from which it is constructed can have synonyms. However, a given pathname cannot lead to more than one segment.

2.3. Links

There can exist in a directory a second type of entry in addition to the branch entry. This is called a "link entry" or a "link". A link entry has a name just as branch entry does and like a branch is used to access a segment or its attributes. A link contains no attributes but only the "pathname" of another entry. In Figure 1, the dotted line labeled L is an example of a link. The entryname of the link is L. Loops such as might be generated by two links which reference each other should not be allowed in the directory hierarchy.

Links are addressed by pathname just as branches are. Referencing the link, > A > B > C > L, in Figure 1, will cause accessing of the segment > D > E > X, as that pathname is recorded in the link.

3. DESIGN CONSIDERATIONS

A directory is needed as a place to look up the addresses of other segments. Once a directory exists, there are other advantages to be gained from it. The directory is a convenient place to store the access rights of a user to a segment so they may be checked at the same time that the address of the segment is located. It may also be useful to reference the attributes of a segment without necessarily accessing the segment body, e.g., to find the length of a segment. For these reasons, all of the attributes of a segment are collected into a list and the lists are stored together in a directory. Why then have more than one directory?

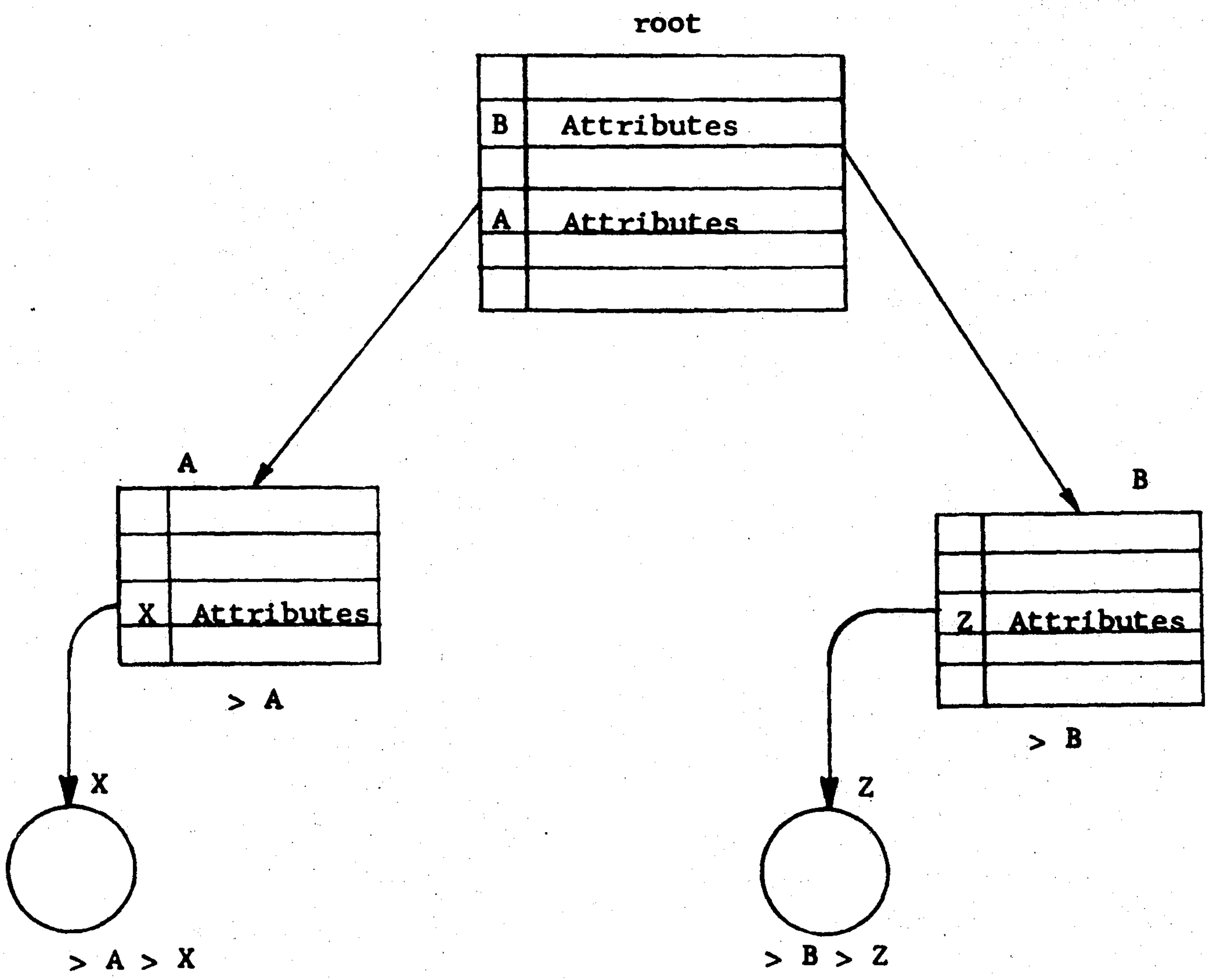


Figure 2. Directories and Attributes

The most important reason for having multiple directories is to avoid the problem of naming conflicts. It is very likely that many users will attempt to use the same names for the different segments they will create. Long and complicated unique names are difficult to remember and inconvenient to look up both for people and computers. If each user is assigned one or more directories then this naming conflict disappears. The key to this simplification is to allow the user to reference the segments of a pre-assigned directory by entryname. Pathnames are constructed by prefixing the directory pathname to the user given entryname and referencing the desired segments via these constructed pathnames.

There are other advantages to be gained with multiple directories. Directories can be used for classification of segments, e.g., all segments of a math library could be accessed through a math library directory, > math-library. Protection is aided since access to directories (and, therefore, complete classes of segments) can be restricted to specific users.

The tree structure carries two advantages. It allows an even better scheme of classification than a linear or limited level structure of directories and it also facilitates an efficient directed search for a given segment in the hierarchy.

4. INTERNAL DIRECTORY AND ENTRY STRUCTURE

Each directory has a small area at its beginning called a header which contains pointers to other areas and items in the directory. There is also in the header a count of the branches in the directory. The header is followed by an array of entries.

Figure 3 shows a directory with a branch entry and a link displayed in some detail. All entries contain the following:

a name list pointer,
a unique identifier and
a branch or link switch.

If the entry is a link, it contains a pathname and no attributes. If the entry is a branch it also contains

an access list pointer,
a segment map,
segment length,
an active switch,
a PT-ASTE pointer and
a directory flag.

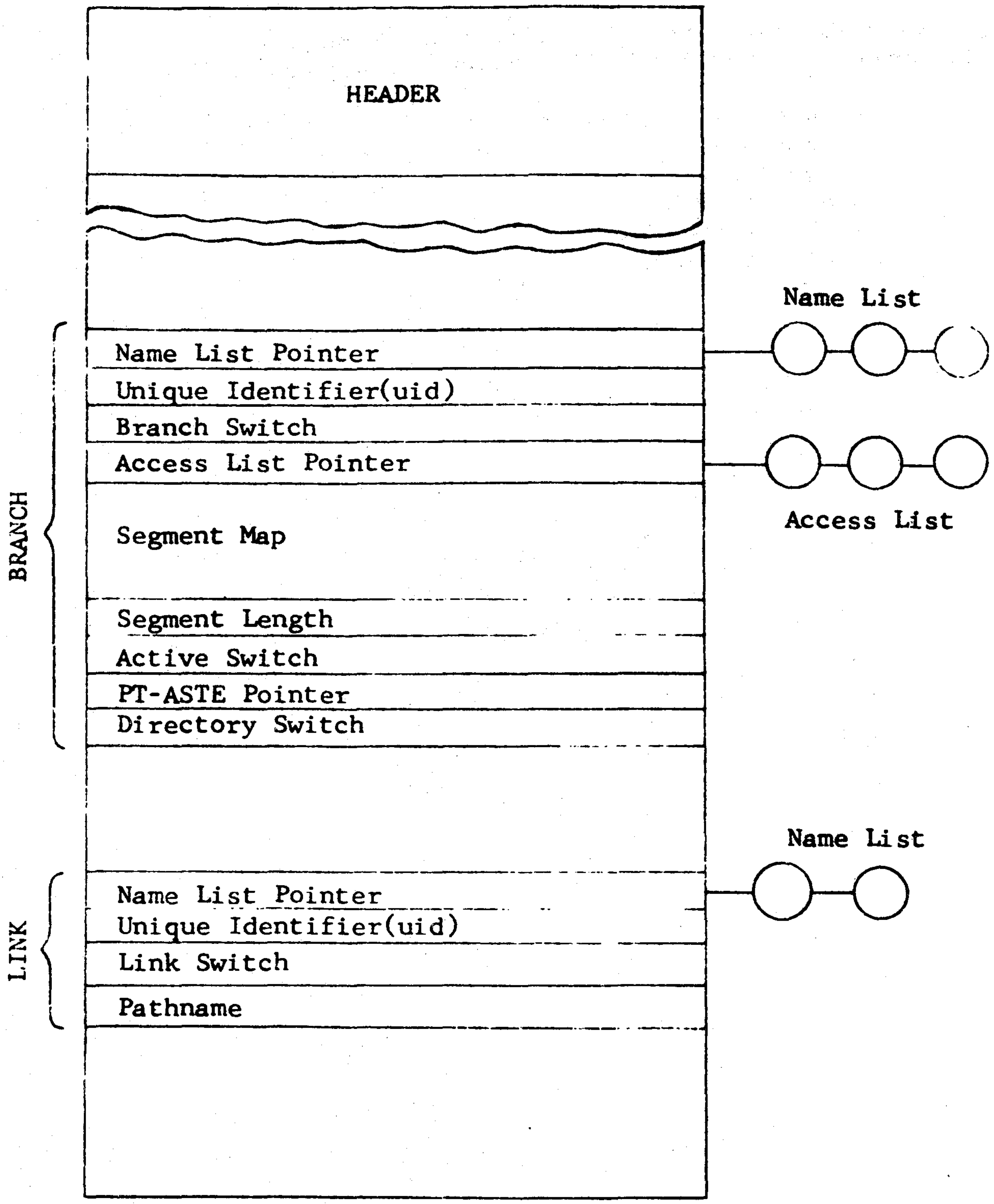


Figure 3 - Structure of a Directory

4.1. The Name List Pointer

A branch or link entry can have several names. These are stored in a threaded list. Supervisor primitives exist to add names to the list and to delete names from it. The name list pointer is a pointer to the threaded list of names of the entry.

4.2. The Unique Identifier

The unique identifier, uid, is permanently attached to the entry. It cannot be changed or destroyed while the entry is in use (contains valid branch or link information). All uid's are constructed in part from numbers representing the date and an instant during the creation of the entry, so that a uid can never be repeated. The uniqueness of the uid for each entry in the entire directory hierarchy is thus guaranteed. The uid is used to assure that the correct segment is being accessed when a segment is referenced or activated. Its use will be described in more detail in Chapters 4 and 5.

4.3. The Branch or Link Switch

The branch or link switch is used to tell the search modules the kind of information to be found in the entry.

4.4. Access List Pointer

The access list pointer is a pointer to a threaded list of user names and associated access rights. As an example, user Tom may have the right to execute the contents of a segment and to modify the contents of that segment while user Frank may only be permitted to execute the contents of the segment. The access list contains the names of all users permitted to access the segment, and is described in detail in a companion paper, "Access Control to the Multics Virtual Memory".

4.5. The Segment Map

As the name indicates, the segment map gives the address of the segment in secondary storage. The segment map consists of two parts, the device indentifier, did, and a page address list. The did is a number which uniquely identifies the particular drum, disc or other device on which the segment resides. For each page up to a maximum of sixty four, there is an address in the page address list which locates that page on the device. For unused pages, there is an unassigned address. The order of the addresses in the address list corresponds to the order of the pages in a segment.

4.6. The Segment Length

The segment length is given in pages. It is one plus the number of the highest page accessed counting from zero. There is a maximum of sixty-four pages for any segment.

4.7. The Active Switch

The active switch, when ON, indicates that some of the segment attributes are to be found in the page table, PT, and its associated active segment table entry, ASTE. The segment may be in core or secondary storage or partly in both when this switch is ON. It is always in secondary storage when the active switch is OFF.

4.8. The PT-ASTE Pointer

The PT-ASTE pointer is a pointer to the location of those attributes of the segment in the page and active segment tables. It is only valid when the active switch is ON. One of the attributes found in the active segment table at that time is the segment map.

4.9. The Directory Switch

Finally, the directory switch indicates whether the segment is a directory or not. This information is used when deleting segments and will also be needed for segment handling which is explained in later chapters.

Other attributes are found in the branch. Some of these will be introduced and explained where needed in later chapters.

5. SEGMENT CREATION

A segment is created by establishing for it a branch in a directory. A module is called with the arguments segment name (pathname of the segment to be created), access to the segment and directory switch. The parent directory of the segment to be created is accessed using the pathname argument with the final entryname truncated from it.

To create a segment, information must be written into its parent directory. This requires that a hardware address be assigned to that directory and, therefore, that the directory be assigned a segment number. The assignment of a segment number is called "making a segment known to a process" and is described in Chapter 4. Subsequent addressing is by segment number and the segment control and page control modules handle the problems of activating the segment and bringing its pages into core.

When the parent directory has been accessed in this manner, a free entry is found in it. The entryname to be given this new branch is checked to make sure that it is not already in use in the parent directory. Space is allotted to the name and access lists and they are moved into their allotted places. Pointers to the name and access lists are placed in the entry. A uid is created for the new branch by a special subroutine and the uid is placed in the branch. The segment map is initialized by assigning to the segment a device identifier and setting all of the page addresses to unassigned. Page control will assign addresses to the pages as they are referenced. The branch count of the parent directory is then incremented by one.

At this point all that need be done to create a non-directory segment has been completed. A test is made to see if a directory is being created. If so, the directory being created is assigned a segment number and accessed. The necessary pointers are placed in its header and its branch count is set to zero. Creation of a directory segment is now complete.

An error return with an appropriate comment would have been executed if a free entry had not been found or the entryname had already been in use or the name and access list area had been full. No segment or entry would have been created in any of those cases.

6. SEGMENT DELETION

As in creation, segment deletion is for the most part deletion of a branch entry. However, before ε segment may be deleted, several checks must be completed.

The parent directory of the segment to be deleted is accessed as in segment creation. The directory switch is tested to see if the segment to be deleted is a directory. If so, it is accessed and its branch count is checked. A directory cannot be deleted if any branches remain in it since that would break the path to all of its inferior segments. If there are no branches in the directory to be deleted then the execution continues as if it were a non-directory segment.

When the directory switch is tested and a non-directory segment is to be deleted, then the active switch in the branch to be deleted is checked. The segment cannot be deleted while this switch is on for that would leave traces of the segment in core and possibly even in other processes. Therefore, segment control is called to deactivate it.

Deactivation removes all traces of the segment from core, in particular from any descriptor segments and from the page table and active segment table entry assigned to this segment. (This forces any subsequent reference of the segment by any process to execute a segment fault thus referencing the branch and seeing that it has been deleted.) Page Control is called to free all secondary storage used by the segment to be deleted. The branch itself is deleted by zeroing its uid. Finally, the branch count of the parent directory is decremented by one and the segment deletion is complete.

Chapter 4

MAKING A SEGMENT KNOWN TO A PROCESS

1. INTRODUCTION

Segments in Multics are identified system-wide, to all users and processes, by their pathnames. However, the hardware references segments by numbers called segment numbers. Therefore, during execution a segment number must be associated with each segment. The segment number associated with any particular segment may differ from one process to another. A segment is said to be "known to a process" (or simply "known") while at least one of its pathnames is associated with a segment number and this association is recorded in a per process segment called the Known Segment Table.

A segment is "unknown to a process" or "unknown" until it has been made known and can again be made unknown to a process by terminating it, that is, by erasing the record of the pathname-segment number association from the Known Segment Table, KST. This breaks the association between pathname and segment number since the record in the KST is the only record of that association.

This chapter describes the way in which segments are made known to a process and the way in which they are made unknown.

2. DATA BASES

There are two major data bases involved in making a segment known. These are the directory hierarchy and the Known Segment Table, KST. Directory structure and contents are discussed in Chapter 3. The KST is discussed in this section.

The KST is a segment with an array of KST entries. Figure 1 is a diagram of the KST with one KST entry, KSTE, displayed in detail. A KSTE contains:

a name list pointer
a segment number
a unique identifier, uid
a branch pointer
a directory switch
an inferior count
and other information.

The header shown in Figure 1 contains housekeeping information pertinent to the KST such as a pointer to the next free KSTE. The various elements in the KSTE are explained below.

2.1. The Name List Pointer

The name list pointer is a pointer to a threaded list of pathnames. The pathnames are all the different synonyms which have so far been used by this process to refer to the same segment, the one associated with this KSTE.

2.2. The Segment Number

Note that there is no segment number in a KSTE. The index of a KSTE in the KSTE array is the segment number associated with that KSTE. It is, therefore, the segment number of the segment whose name is pointed to by the name list pointer. In Figure 1, "s" is the segment number of the segment whose names are in the pathname list of the expanded KSTE.

2.3. The Unique Identifier, uid

The uid is copied from the branch of the segment when the segment is made known. The uid uniquely identifies a branch and is described in Chapter 3.

2.4. The Branch Pointer

When a segment is made known a pointer to its branch is placed in its KSTE. The presence of the branch pointer in the KSTE is necessary since the name of the segment could be changed while the segment is known. It also allows the segment-fault handler easy access to the attributes of a segment without having to repeat the search for the branch.

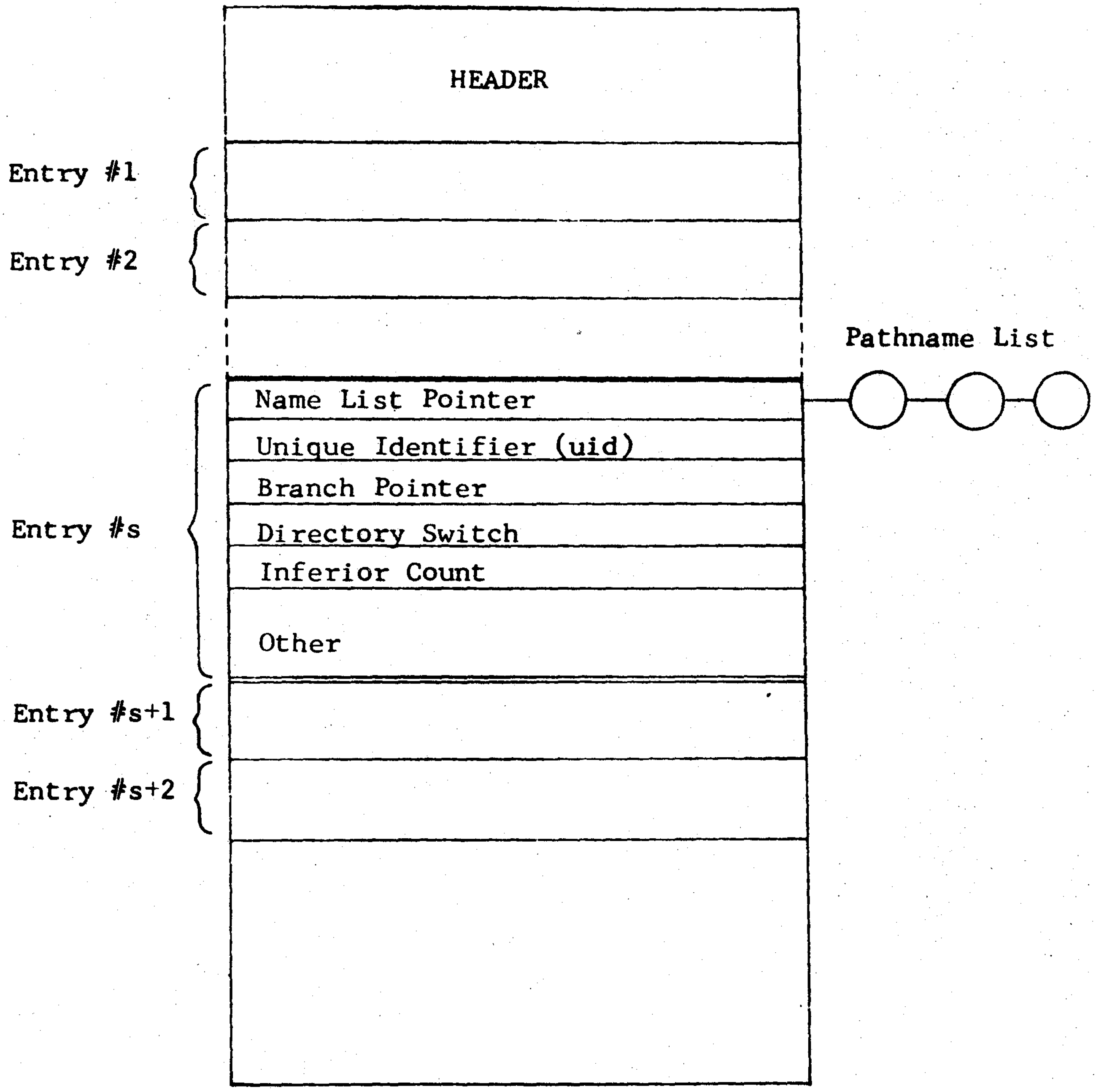


Figure 1. Structure of the Known Segment Table (KST)

2.5. The Directory Switch

The directory switch simply tells whether a segment is a directory or not. It is present because of a special rule regarding the handling of directories when making them unknown.

2.6. The Inferior Count

The inferior count is used for directory segments. It is a count of the number of immediately inferior or daughter segments known in the process to which this KST belongs.

2.7. Other Information

There is other information in a KSTE which is put there for use in access control, for example, but which is not pertinent to this discussion. No further mention will be made of it.

3. MAKING A SEGMENT KNOWN

The procedure executed in order to make a segment known is illustrated by the flow chart in Figure 2. Entry is through a call to MAKE-KNOWN. The pathname of the segment to be made known is passed to MAKE-KNOWN and the segment number and a code are returned.

The KST is searched for the pathname and if the pathname is found, MAKE-KNOWN returns a segment number. The third argument, code, is set to inform the caller whether or not the segment just made known is a directory directory. If the pathname passed to MAKE-KNOWN was not found in the KST, then it is tested to determine if it is the root directory pathname. This step is very important as it assures an end to the recursive loop which is described below.

If the pathname is not that of the root directory, then the pathname is parsed and is broken into two parts, a new pathname, and an entry name. The new pathname is the pathname of the directory in which the new entry name can be found. For example, if > A > B > C > D were the original pathname, then the parse would yield the new pathname > A > B > C and the entry name D. MAKE-KNOWN is then called recursively to make the new directory pathname known. This recursive loop is executed until a directory pathname is found in the KST or until the root directory is encountered and made known.

A call to make the root directory known always terminates the recursive loop. Since all pathnames begin with the root pathname it is assured that the recursive loop will always be terminated.

When MAKE-KNOWN returns from a recursive call the code argument is checked to make sure that a directory was found. The directory is searched for the entryname separated from a previous pathname by the parse.

An example is useful at this point. Assume that the segment with pathname > A > B > C is to be made known and that MAKE-KNOWN has been called twice (once recursively) for this segment. On the initial call the pathname > A > B > C was passed to MAKE-KNOWN and on the second call (first recursive call) > A > B was passed to it. Assume that directory segment > A > B was found to be known, then upon return from the last call (the recursive call) the code argument is tested to see if > A > B is a directory. If so, then entry C is looked up in > A > B. We now return to the description of MAKE-KNOWN.

If the entryname is found in the directory whose segment number was just returned, then the entry is tested to determine if it is a branch or a link. If it is a branch, then its uid is looked up in the KST to make sure that the segment is not already known by some other name. If the segment is already known by some other name then the new name is added to its pathname list, the segment number is returned and the code argument is set from the directory switch in its KSTE. If the segment is not known, then a free KSTE is found. The pathname is allocated space and placed in that space. The name list pointer is set to point at the pathname, the uid and the directory switch are copied from the branch into the KSTE and the branch pointer is set in the KSTE. The code argument is set from the directory switch in the branch and the KSTE inferior count of the parent directory is incremented by one. Finally, MAKE-KNOWN returns to its caller (itself or some external caller).

When there is a call to make the root known a special procedure in MAKE-KNOWN is used. Special data is entered into the KSTE for the root without searching the directory hierarchy or attempting to find a branch for it. Its segment number and code are returned in the normal manner.

When a link entry is found, then MAKE-KNOWN is called with the pathname found in the link and the flow continues in the normal manner.

If an error is encountered such as not finding a directory on return from a recursive call to MAKE-KNOWN, then an error code is set into the code argument and a null segment number is returned.

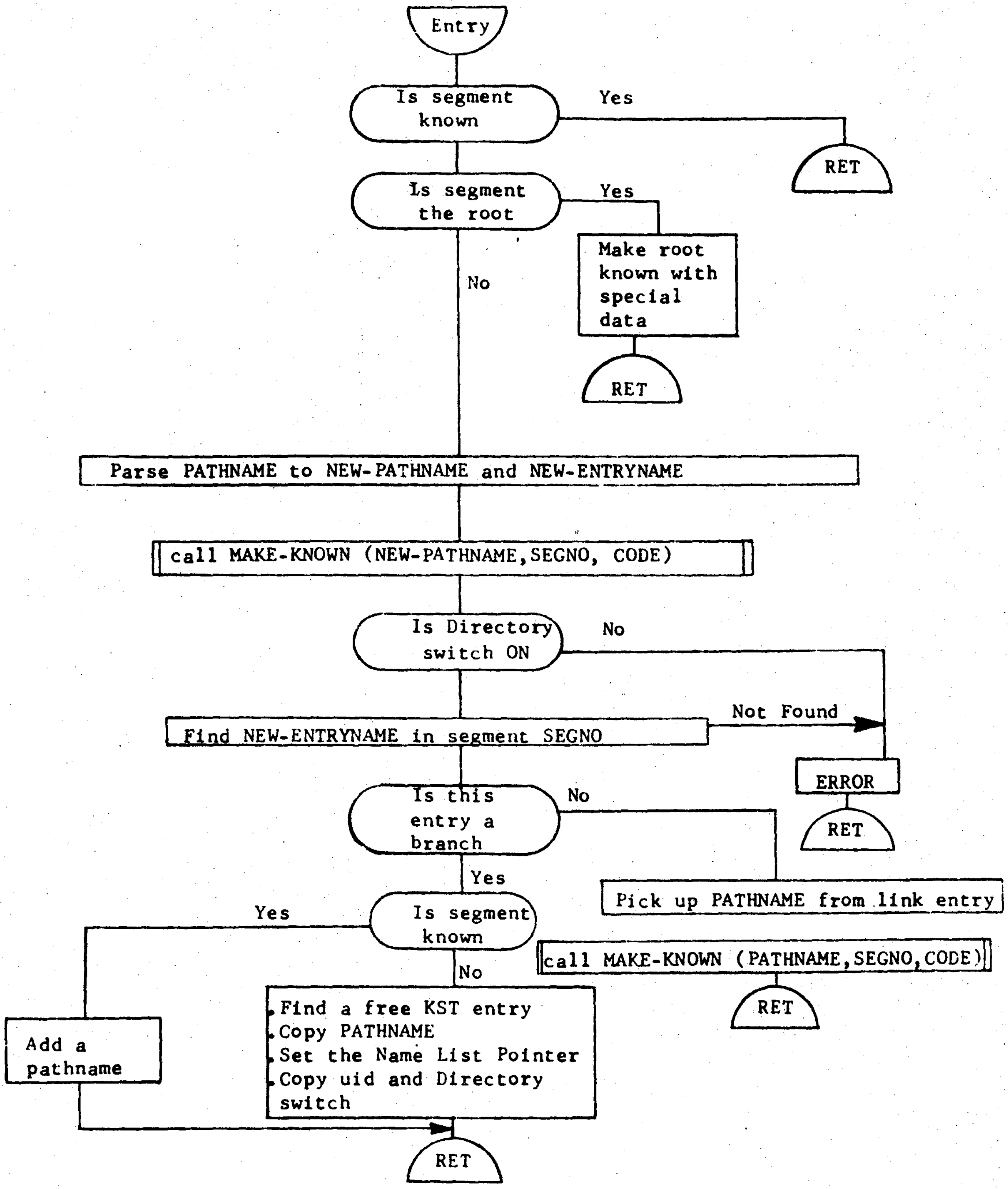

Entry
Is segment known
Yes
RET
Is segment the root
Yes
Make root known with special data
RET
No
Parse PATHNAME to NEW-PATHNAME and NEW-ENTRYNAME
call MAKE-KNOWN (NEW-PATHNAME,SEGNO, CODE)
Is Directory switch ON
No
Find NEW-ENTRYNAME in segment SEGNO
Not Found
ERROR
RET
Is this entry a branch
No
Yes
Pick up PATHNAME from link entry
call MAKE-KNOWN (PATHNAME,SEGNO,CODE)
RET
Is segment known
Yes
No
Add a pathname
.Find a free KST entry
.Copy PATHNAME
.Set the Name List Pointer
.Copy uid and Directory switch
RET

Figure 2. MAKE-KNOWN

4. MAKING UNKNOWN

Normally, a user will experience no difficulty because of the limited size of the KST. However, since there are many more segments in the directory hierarchy than there are KSTE's, a user might wish to free a KSTE (making a segment unknown) so that it can be reused. In any case, certain precautions must be taken before a segment can be made unknown.

First, the directory switch in the KSTE is checked. If the segment to be made unknown is a directory then its inferior count is checked. A directory cannot be made unknown if any of its inferior segments are known. This convention is stated in Chapter 2. It arises from our desire to be able to take segment faults on segments used in the Multics supervisor and not to distinguish between supervisor and user segments.

Second, segment control must be notified that a segment is being made unknown since the KST is prepared for and used by segment control. This is done by a special call to segment control. Upon receiving this call, segment control will disconnect the SDW associated with this segment in this process (see Chapter 5). When segment control has been notified the segment can be made unknown by freeing the area where its name(s) was stored and threading the KSTE onto a free KSTE list. The parent directory's KSTE inferior count is then decremented by one and the operation of making segment unknown is complete.

Finally, a few words must be said about the danger of making a segment unknown and reusing its KSTE. Addresses are prepared using segment numbers. All of these addresses cannot be found when a KSTE is to be freed. If such an address is used after the KSTE has been reused, it will cause information in the corresponding KSTE to be used without further checking. Incorrect segment referencing would result. Therefore, a segment should not be made unknown and its KSTE reused unless it is assured that no address which references that segment will be used again during the existence of the process.

5. INITIAL REQUIREMENTS

The question may be asked, "how much apparatus is required to make a segment known for the first time in a process?"

The data bases used are the directory hierarchy and the KST. The procedures used are MAKE-KNOWN and the procedures called by it. It has been shown that all directories including the root directory may be made known. However, the root directory requires special code. The KST must already have a segment number in order to make a segment known so the KST cannot be made known in this way. Some of the modules called by MAKE-KNOWN could possibly themselves be made known, however, special codes would be necessary to do this. Therefore, MAKE-KNOWN and all of the procedures used by it as well as the KST for a process are assigned segment numbers before the process begins executing as a part of process initialization.

6. OTHER MULTICS CONSIDERATIONS

The fact that some segments must have segment numbers before MAKE-KNOWN can be executed gives a clue to the Multics implementation of segment numbers. There is a group of segments in the Multics supervisor which must have segment numbers assigned in a process before the process can begin execution. These are called hardcore segments. They have no KSTE's. The actual segment numbers assigned to segments when they are made known are the KSTE index plus the highest segment number assigned to a hardcore segment.

Chapter 5

SEGMENT FAULT HANDLING

1. INTRODUCTION

In the Multics Operating System, each process address space is divided into 64K-word items called segments. A segment enters a process address space by being "made known to the Process" (see Chapter 4). In the course of being made known, a segment has a per-process segment number assigned to it. A correspondence is established between this segment number and the segment's pathname and attributes in a per-process table called the "Known Segment Table". Once in a process address space, a segment may be referred to by segment number.

Whenever a process references memory, the 645 hardware references "per process" and "per system" registers. The "per process" information in the hardware accessing path is recorded in a Descriptor Segment which contains one word, called a Segment Descriptor Word (SDW), per segment number. The function of the Nth SDW is to point to the Page Table (see Chapter 6) of the segment which is known to the process as segment #N and to specify the process access rights with respect to that segment.

The Page Table of a segment (and various other data required by the paging mechanism of Multics) must be stored in core. Since there are more segments in Multics than places in core for Page Tables, not all segments can have Page Tables at the same time. When a segment has a Page Table, the segment is called ACTIVE. At other times it is called INACTIVE.

An SDW can, of course, contain the address of a segment's Page Table only if the segment is active. If an SDW contains the address of the segment's Page Table and specifies the process access and the segment's length, then the SDW is called CONNECTED; otherwise, it is called FAULTED or DISCONNECTED. A faulted SDW in fact contains a bit pattern which, when encountered by the 645 addressing hardware, causes the process to "take a Segment Fault" thereby invoking the Segment Fault Handler. See Figure 1.

In view of the above, we may say that:

> The function of the Segment Fault Handler is to provide the process with the illusion that all segments known to it are active and all SDW's corresponding to known segments are connected; in short, to render all known segments directly accessible by segment number.

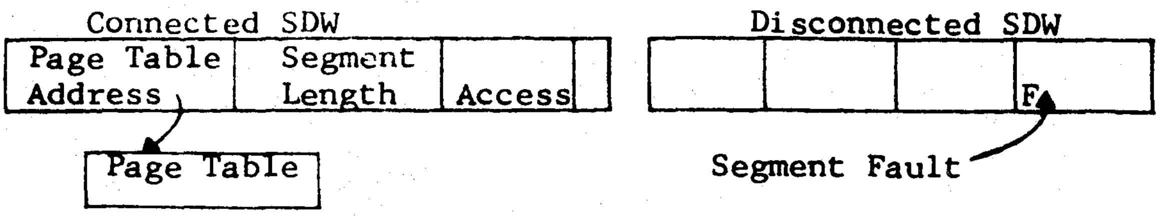


Figure 1. Connected and Disconnected SDW's

2. PREVIEW OF THE SEGMENT FAULT HANDLER (SFH)

2.1. Procedure

A segment fault occurs when a process attempts to access a "target" segment via a faulted SDW. The Segment Fault Handler (SFH), called to repair the faulted SDW, must obtain the address of the Page Table for the "target" segment as well as the process access rights to the segment and store the information in the SDW.

To do this, the SFH must:

a. Check the validity of the segment number given to the SFH. It may have been incorrectly generated.

b. Use the segment number of the "target" segment (which the SFH is given) to find a pointer to the segment's branch. This "branch pointer" was put into the segment's entry in the process Known Segment Table (KST) when the segment was made known. (See Chapter 4).

c. Check the branch to see if it in fact corresponds to the "target" segment. This check is necessary due to the dynamic nature of the File System in which segments can, at any time, be created and destroyed or moved from one directory to another. In checking the branch, a unique identifier (UID) is used which was stored in the segment's KST entry when the segment was made known.

d. Look in the branch, which contains the segment's attributes, to find the process access to the segment and to locate the segment's Page Table.

e. Repair the SDW and return.

2.2. Data: Active Segment Table (AST)

We have stated that Page Tables and other data describing active segments must be stored in core, a requirement imposed by the Page Fault Handler (see Chapter 6). Use is, therefore, made of a system-wide table, the Active Segment Table (AST), which resides permanently in core. The AST is a linear array containing one entry per active segment. An active segment's AST Entry (ASTE) contains the segment's Page Table and other paging data and, as we shall see, other data needed by the Segment Fault Handler.

Steps (a) through (e) above and the description of the AST lead to the picture of the data structures used by the Segment Fault Handler and the relations between them shown in Figure 2.

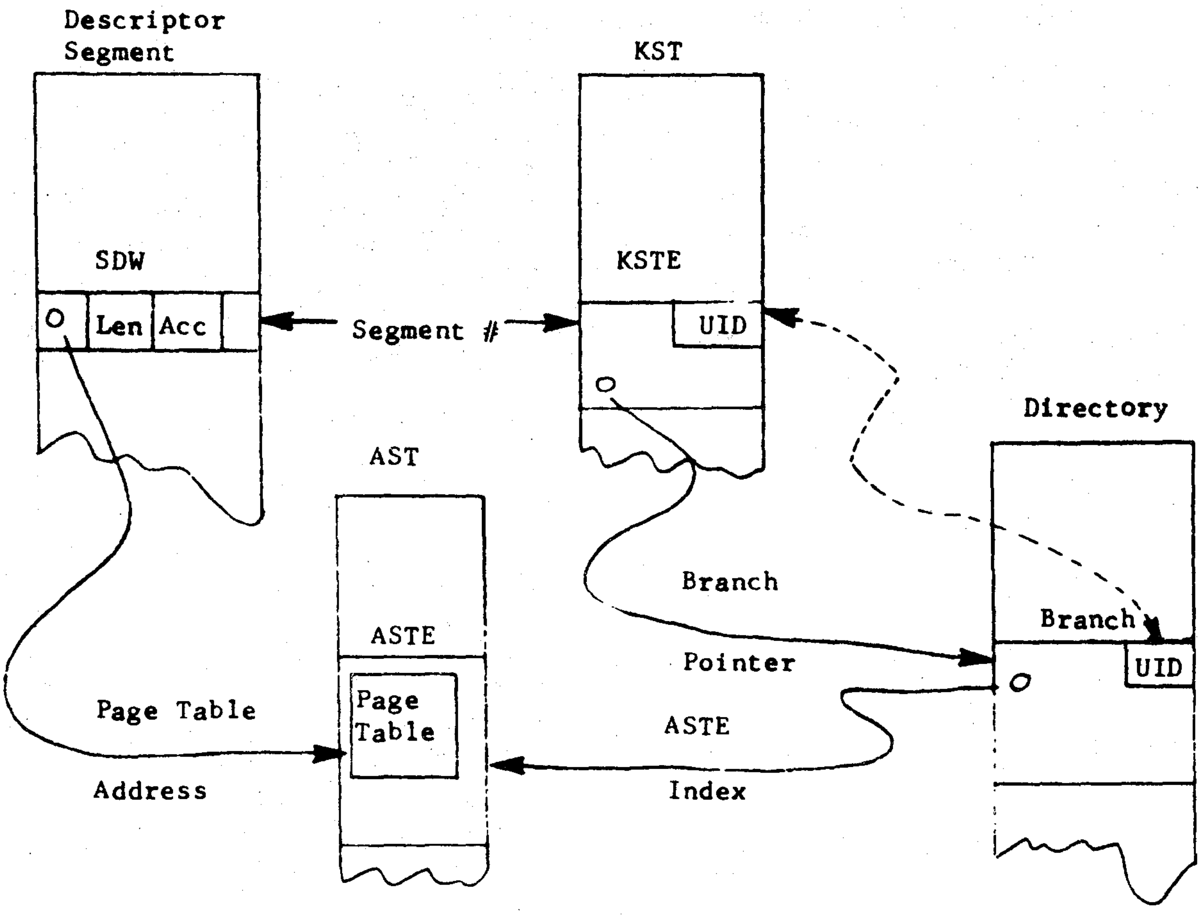


Figure 2. Principal Data Structures Used by the Segment Fault Handler

2.3. Program of Exposition

In order to make segment fault handling more easily comprehensible, it will be useful to discuss the procedures for handling segment faults in three sections corresponding to three successively more likely assumptions about the state of activity of the segments of the system.

These assumptions are:

1. All segments are active simultaneously.
2. All segments can be active simultaneously.
3. All segments cannot be active simultaneously.

3. SEGMENT FAULT HANDLING WHEN ALL SEGMENTS ARE ACTIVE

Let us examine the very unlikely case in which all segments are active. The Segment Fault Handler (SFH) begins by finding the "target" segment's KST entry and branch as explained in Section 2. Two kinds of errors may happen. First, if no segment is "known" by the faulting segment number, then no KST entry exists. This error may happen, for example, if a memory reference is made via a randomly generated segment number. This error is detected through structural details of the KST which do not interest us here.

Second, as mentioned in Section 2, the branch pointer in the KST entry will be incorrect if the "target" segment has been destroyed since this process made it known. To check the correctness of the branch pointer, the SFH compares the unique identifier (UID) in the branch with the one in the KST entry. If they are the same, then the branch pointer is alright; otherwise, the "target" segment has been destroyed and the segment fault cannot be satisfied.

If the branch pointer is valid, the SFH gets the process access rights to "target" segment and the index of the "target" segment's ASTE from the branch. (The branch of an active segment must, of course, contain a pointer to the segment's ASTE. The ASTE index is that pointer.) The address of the "target" segment's Page Table can be calculated easily from the ASTE index. The length of the segment (in pages) can be obtained from the ASTE.

With the table address, the segment length, and the access information, the SFH has enough information to correct the faulted SDW. The SDW is corrected and the SFH returns.

4. SEGMENT ACTIVATION

We wish now to see how segment fault handling must differ if it is possible that the "target" segment may not be active. We assume that there is room in the AST to make an ASTE for any segment. Two new data structures must be introduced.

First, a new piece of information must be added to the branch: an active switch. This switch indicates whether or not the segment is active; i.e., whether the ASTE index in the branch may be used. Second, a new data structure must be added to the AST - a list of available entries.

This structure is called the AST free list. In this section, we assume that the AST free list is never exhausted.

Now let us look at segment fault handling. Once the branch has been accessed, the SFH must inspect the "active switch". If the segment is active, the processing is as described above. If the segment is not active, it is necessary to activate it, that is, to:

1. Find an entry in the AST free list. Remove it from the AST free list.

2. Set the ASTE index in the branch to point to this entry.

3. Copy the paging data from the branch to the ASTE. Initialize the Page Table in the ASTE.

4. Set the branch's "active switch" on.

After activating the segment, the SFH proceeds as before to repair the faulted SDW.

5. SEGMENT DEACTIVATION

Let us now consider the real case in which the size of the AST limits the number of segments which can be active simultaneously. The discussion of this case is sufficiently long that we will divide it into two parts. In this section we will introduce the pieces of the design. In the following section we will put the pieces together.

The assumption that all segments cannot be active simultaneously implies that it may sometimes be necessary to activate a segment at a time when the AST is "full"; i.e., there are no ASTE's in the AST free list. When this happens it becomes necessary to:

a. Choose an active segment to be deactivated.
b. Deactivate this segment.
c. Return the ASTE of the deactivated segment to the AST free list.

Let us defer discussion of how an active segment is chosen for deactivation until the mechanism of deactivation has itself been discussed.

Deactivation of a segment may be characterized as doing all things necessary to disassociate the segment's ASTE from the segment, thus permitting the ASTE to be used to activate another segment. It is important to note at this point that the segment being deactivated is not necessarily "known" to the process performing the deactivation. Many of the details of the design arise from this fact. Section 9 discusses the matter further.

Deactivation is done in the following three steps:

i page removal - forcing the segment's pages out of core.

ii disconnection - seeing that all SDW's associated with this segment are faulted;

iii restoring the branch - moving back to the branch the (presumably altered) values of those attributes of the segment which were moved to the ASTE when the segment was activated.

5.1. Page Removal

A Page Table for a segment can only exist when the segment is active; when the segment is deactivated, the Page Table is destroyed. Thus, one function of deactivation is to remove the segment's pages from core and to inhibit the Page Fault Handler from bringing any of its pages into core during deactivation. To accomplish this, the SFH calls a special entry of the paging module which removes the segment's remaining pages from core and causes page faults (in other processes) on pages of this segment to be transformed into segment faults on this segment.

5.2. Disconnection

Disconnection obviously amounts to faulting all SDW's connected to the segment at the time of deactivation. There may, of course, be many other SDW's associated with the segment, but as they are not connected they must (already) be faulted. In order to make disconnection possible, a data structure must be associated with each segment which lists all the SDW's connected to the segment. Since this connection list can only be non-empty when the segment is active, it can be stored in the segment's ASTE and need never appear in the segment's branch. When a segment is activated, the connection list is empty. Whenever an SDW is connected to a segment, a pointer to the SDW is added to the segment's connection list. When the segment is deactivated, each of the SDW's on the associated connection list is faulted.

5.3. Restoring the Branch

After page removal and disconnection have been done, all that remains of deactivation is to restore the segment's branch. It is necessary to move back to the segment's branch those attributes which were moved from the branch to the ASTE when the segment was activated (which may have changed during the period of activation) and to reset the "active switch". In order to manipulate the segment's branch in this way, the SFH makes use of another data element in the ASTE of each segment, a pointer to the segment's branch. This branch pointer must be stored in the ASTE when the segment is activated so that it can be used again when the segment is deactivated.

5.4. A Note on Certain Necessarily Active Segments

The reader can easily convince himself that the main path through the SFH (excluding the deactivation path) can only be viable if certain segments are perpetually active: for example, all segments required by the Page Fault Handler, the SFH procedure itself, the per-process KST segment, etc. To permit such segments to be held active, another item is added to the ASTE, the entry hold switch, which may be set to prevent the deactivation of the segment. We will consider these matters further in Section 8 on "Recursion and Initialization." Here we wish to consider a different problem.

During deactivation of a segment the SFH must access all the descriptor segments which contain SDW's connected to the segment being deactivated; it must fault these SDW's. The SFH must also access the segment's branch. In order to assure that these segments may be accessed easily, the following rules are established:

Rule 1 - A Descriptor Segment containing a connected SDW must be active.

Rule 2 - A Directory Segment with an active daughter segment must be active.

Rule 1 may be enforced by use of the entry hold switch. The enforcing of Rule 2 is more complicated. A new data element is associated with each active segment, the <u>inferior count</u>. This is a count of the active daughter segments of the given segment. (For non-directory segments the inferior count is necessarily zero.) When a segment is activated, its own inferior count is set to zero and one is added to its parent's inferior count. When a segment is deactivated, one is subtracted from its parent's inferior count. Needless to say, no segment is chosen for deactivation whose inferior count is non-zero. To enable the deactivation machinery to access the parent directory's inferior count, each ASTE also contains another data element: the <u>parent's ASTE index</u>.

5.5. <u>The Deactivation Algorithm</u>

We have now presented all of the data structures needed to permit deactivation of a segment. Let us now discuss the "deactivation algorithm" which must choose the segment to be deactivated. To facilitate deactivation, another list is introduced, the <u>AST used list</u>, which contains all of the ASTE's corresponding to active segments.

In order to keep deactivation economical, the deactivation algorithm chooses segments with as few pages in core as possible. In practice, it is very often able to choose a segment with <u>no</u> pages in core. The algorithm essentially passes through the AST used list as many times as necessary; on the Nth pass it looks for a segment with fewer than N pages in core whose inferior count and entry hold switch are both zero. When such a segment is found, the search stops and the segment is deactivated.

6. FLOW OF THE COMPLETE SEGMENT FAULT HANDLER

Correct Segment Fault

1. Given segment number, find KST entry.
   Check validity of KST entry.
   In KST entry, obtain pointer to the branch.
   Check validity of the branch pointer by comparing the KST entry's and the branch's version of the unique identifier.

2. If the "active switch" is set, go to Step 8.

Activate the Segment

3. Inspect the free list. If there is a free ASTE, then go to Step 7.

Obtain a Free Entry

Choose a Segment to be Deactivated

4. Pass through the AST used list as many times as necessary; on the Nth pass look for a segment having fewer than N pages in core with inferior count and entry hold switch equal to zero. When a segment is found in this way, go on to Step 5.

Deactivate the Segment

5. Remove the pages of the segment from core by a call to the paging module.
   Disconnect all of the SDW's listed in the segment's connection list.
   Use the ASTE's branch pointer to move the segment's attributes back to the segment's branch.
   Reset the "active switch" in the branch.
   Use the parent's ASTE index in the ASTE to locate the parent's inferior count, from which subtract one.

6. Remove this entry from the AST used list.
   Put it in the AST free list.

Activate the Segment

7. Move the ASTE from the AST free list to the AST used list.
   Move certain attributes from the branch to the ASTE.
   Set the branch's ASTE index and set the "active switch".
   Initialize the Page Table in the ASTE.
   Set the segment's inferior count to zero.
   Set the segment's entry hold switch to zero.
   Find the parent's ASTE; store its index in the segment's ASTE.
   Add one to the parent's inferior count.

Connect the SDW

8. Get the page table address and segment length from the ASTE.
   Get the access field for the SDW by inspecting the Access Control List in the branch.
   Store the page table address, segment length, and access field in the SDW. Reset the Segment Fault Flag.

9. Return.

7. DATA USED IN SEGMENT FAULT HANDLING

7.1. AST (Active Segment Table)

The AST consists of a Header and a linearly indexed array of AST Entries of which there is one per active segment. The AST Header contains:

a. The head of the AST free list.
b. The head of the AST used list.
c. The branch of the root directory.

Each AST Entry contains, in addition to forward and backward pointers used in threading the entries into the various lists,

a. Paging data
   1. The PAGE TABLE.
   2. The SEGMENT MAP.
   3. Various other paging data.

b. Data used in choosing a segment for deactivation.
   1. The INFERIOR COUNT. If the segment is a directory segment, the inferior count is the number of its active daughter segments. Otherwise it is zero.

   2. The NUMBER OF PAGES IN CORE (a counter maintained by the paging mechanism).

   3. The ENTRY HOLD SWITCH. When set, this switch makes the segment ineligible for deactivation.

c. Data required to deactivate the segment.
   1. BRANCH-POINTER
      i ASTE index of the segment's parent directory.
      ii Offset of the segment's branch within the parent directory.

   2. CONNECTION LIST, the list of SDW's which point to the Page Table of this segment and which must be faulted before the segment can be deactivated. Each element of the list consists of:

      i ASTE index of the descriptor segment containing the SDW.
      ii Offset within that descriptor segment of the SDW (i.e., the segment number within that process of this segment).

7.2. SDW (Segment Descriptor Word)

A Descriptor Segment is a linear array whose entries are words called SDW's (Segment Descriptor Words). The index of an SDW in a process Descriptor Segment in the segment number, in that process, of the segment associated with the SDW. An SDW consists of:

a. The ACCESS FIELD
   1. If zero, this field causes a segment fault to occur upon an attempted access via this SDW. In this case, items (b) and (c) below are meaningless.

2. If non-zero, this field indicates the accessing permission that this process has toward the segment associated with this SDW (e.g., "read", "write", or "execute" permission).

b. The PAGE TABLE ADDRESS of the associated segment.

c. The SEGMENT LENGTH (in pages) of the associated segment.

7.3. KST (Known Segment Table) - See Chapter 4.
7.4. Branch - See Chapter 3.
7.5. Page Table, Segment Map - See Chapter 6.

8. RECURSION AND INITIALIZATION

The phrase "recursive segment fault" refers to a segment fault taken by the SFH. In Section 5, we asked the reader to believe that segment faults must not be permitted to occur on certain segments; e.g., the AST. In this section, we shall show why this is so and how such segment faults can be avoided. We shall then consider the recursive segment faults which can be permitted and will conclude with an example.

We may regard each segment fault "taken by the user" as the first in a sequence of segment faults, the rest of which are taken "recursively" by the SFH in handling the first one. It is a design requirement that all such sequences be finite. This implies that the last segment fault in the sequence must be handled entirely by segments which are active and connected in the process handling the faults. Thus, certain segments must be active and connected in a process before the first segment fault is taken.

Which segments must be active? Certainly all segments must be active and connected which are necessarily called in the course of handling every segment fault. In the present design these comprise the AST, KST, SFH procedure and of course all segments required by the paging mechanism (since the SFH references segments which do not reside permanently in core and may take page faults).

Multics initialization must make all of these segments, except the per-process KST, active before the first process is created. Process creation and initialization must create and activate the KST and connect all of these segments before the first segment fault is taken. These segments may be kept active in various ways. Per system segments like the AST and SFH procedure may be kept active by leaving their ASTE's off the AST used list. The per user KST may be kept active by setting the entry hold switch in its ASTE while the process itself is active.

The only segments referenced by the SFH beside those discussed above are directories, the segments which contain branches. The only way to avoid recursive segment faults altogether is to require that every directory, any daughter segment of which is <u>known</u> to any process, be active and appropriately connected. This idea must be rejected as impractical. Hence, recursive segment faults must be reckoned with.

We shall give an example of a sequence of recursive segment faults at the end of this section. We shall show there that all such sequences can be handled if a segment fault on the root directory can be handled. Let us prepare for the example by considering the root directory more closely.

Every segment in Multics except the root directory has a branch in a directory; the branch contains the attributes of the segment. Since no directory is superior to the root, it cannot have a branch in a directory. Nevertheless, if the root is to be accessed, its attributes will have to be recorded somewhere. Since we normally think of a branch as the locus of a segment's attributes, we may provide for the root's accessibility by providing it with a branch. This branch must itself be accessible; since no process can take a segment fault on the AST, it is sufficient that:

> The root directory has a branch which resides in the AST segment.

Whenever the root directory is "made known" in a process, a pointer to this branch is placed in its KST entry. Segment faults on the root can obviously be handled in the normal way whether or not the root is active at the time of the segment fault.

NOTE: In practice, a more efficient (if less straightforward) SFH may be obtained by handling segment faults on the root with special code. If this is done, the "branch" of the root disappears from the AST and, by way of trade-off, the SFH procedure segment gets longer.

We have noted that the only segment fault that the SFH can take while handling a segment fault for a segment is a fault on that segment's parent directory. Thus, every sequence of recursive segment faults corresponds to a path up the directory tree structure toward the root directory. Let us now discuss the canonical example of a sequence of recursive segment faults.

8.1. Example of Recursion

Let us assume that a segment fault is taken for a segment with pathname

root > d1 > d2 >...> dN > seg

Let us further assume that the process SDW's for all the directories "root", "d1", etc., are faulted. Early in handling the fault on "seg", the SFH references the branch of "seg" which causes a segment fault on "dN". The corresponding reference to the branch of "dN" causes a fault on "dN-1" and so it goes until there is a fault on "root". Since the branch of "root" lies in the AST which is active and connected, this segment fault can be handled without another segment fault. With the "root" connected, the SFH can go on with handling the fault on "d1". When "d1" has been connected, "d2" can be handled, and so on. Thus, the only recursive segment fault permitted in the present design is that on a segment's parent; and we have shown that the recursion terminates due to the special treatment of the root directory.

9. SPECIAL ADDRESSING IN DEACTIVATION

9.1. Basic Problem

The deactivation of a segment requires the accessing of a directory (the segment's parent) and of one or more Descriptor Segments (those containing SDW's connected to the segment).

It is very unlikely that these segments are all "known" to the process performing the deactivation. How, then, can the branch pointer and the SDW pointers in the connection list in the ASTE be implemented?

9.2. The Multics Solution

In the present design, the parents of active segments and all descriptor segments connected to active segments are required to be active. This requirement enables the use of the ASTE index as segment specifier for all of the special addresses needed in deactivation. Thus, the branch pointer and SDW pointers are all of the form:

(segment's ASTE index, offset)

Let us look in detail at the trick by which one of these special addresses is used, say the "branch-pointer". A special segment number is reserved in each process for use by the SFH. To access the branch during deactivation, the SFH uses the branch's ASTE index to compute the branch's page table address. This page table address and read and write access permissions are then stored in the SDW which corresponds to the reserved segment number in the deactivating process Descriptor Segment. A pointer using the segment number and the offset given in the "branch-pointer" is then constructed and used to access the branch through the just manufactured SDW.

In effect, the reserved segment number and the corresponding SDW constitute a "window" through which a process can reference a segment not formally known to it.

10. SPECIAL ENTRIES OF THE SEGMENT FAULT HANDLER

There are four functions related to segment fault handling which are accomplished by special entries of the SFH.

10.1. Obtaining a Free ASTE

Some procedures need to be able to obtain a free ASTE in order to activate a segment "by hand". For example, the process creation procedure must make up an active descriptor segment for a new process. A special entry in the SFH is provided which obtains a free ASTE (whether from the AST free list or by deactivation), detaches it from the free list, does not thread it into the AST used list, and returns its index to the caller.

10.2. Setting Faults for All Users of a Segment

Certain procedures wish to fault all SDW's connected to a segment. For example, the Access Control Module will do this if the access control data in the segment's branch has been modified in certain ways. These faults are set by using the part of the deactivation code which disconnects SDW's.

10.3. Deactivating a Segment

Certain procedures must be able to deactivate a segment, for example, the procedure which destroys a segment. Deactivation is accomplished by using the deactivation path in the SFH.

10.4. Disconnecting an SDW from a Segment

Some procedures, for example, the procedure which makes a segment "unknown" to a process, need to be able to disconnect an SDW from a segment. Code in a special entry of the SFH is provided which faults the SDW and removes it from the list of SDW's connected to the segment.

11. CONCLUDING REMARKS

The functions of the Segment Fault Handler (SFH) are:

- to make (the Page Tables of) all segments which are "known" to a process accessible to the process by segment number.

- to multiplex the system's relatively few AST Entries (or, equivalently, Page Tables) among all of the segments "known" to all of the processes executing in the system.

To accomplish these functions, the SFH must establish (activate) and dis-establish (deactivate) the pageability of segments and must establish (connect) and dis-establish (disconnect) the use of such pageability by processes. It is instructive to separate these four functions into two groups.

11.1. Connection and Activation - Process Oriented Functions

The primary tasks in handling a segment fault, connection and activation, are done on a demand basis according to the needs of a process. These functions are performed using only data segments "known" to the process - the per-process KST and Descriptor Segment, the directories superior to the "target" segment, and the AST segment. In this part of handling a segment fault, two types of error may occur. The segment number may (a) not correspond to a segment "known" to the process, or (b) may correspond to a segment which has been destroyed. In either case, the process is trying to access a segment which does not exist and should note the error. For all of these reasons, we say that the connection and activation aspects of segment fault handling are "process oriented functions"; i.e., that the SFH in performing them is acting "for the process".

11.2. Disconnection and Deactivation - System Oriented Functions

The secondary tasks of segment fault handling, disconnection and deactivation, are undertaken by the SFH during activation only when the state of the system demands it, the relevant state of the system being the emptiness of the AST free list. The choosing of the segment to be deactivated is independent of the process in which the SFH is executing. Deactivation and disconnection of a segment require the SFH to access segments not (necessarily) "known" to the process executing the SFH: the parent directory of the segment being deactivated and the Descriptor Segments containing SDW's connected to the segment. The method by which these segments are accessed by the SFH (see Section 9) is clearly independent of the address space of the process in which the SFH is executing.

(The number of directory segments and Descriptor Segments in the system may well exceed 2^{18}. All of these segments are potentially referenceable by any process which is executing in the SFH deactivation path.)

No errors are detected (or caused) during deactivation and disconnection and no per-process errors affect their operation. For these reasons, we say that the deactivation and disconnection aspects of segment fault handling are "system oriented functions", i.e., that the SFH in performing them is acting "for the system".