M0108

# A GUIDE TO MULTICS

# FOR

# SUBSYSTEM WRITERS

## CHAPTER VI

## Segment Management and Directory Structure

Elliott I. Organick

Draft No. 4

March, 1969

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

## TABLE OF CONTENTS

## LIST OF ILLUSTRATIONS

## LIST OF TABLES

# CHAPTER VI

# SEGMENT MANAGEMENT AND DIRECTORY STRUCTURE

## 6.1 INTRODUCTION

The file system plays a central role in the execution of every process. Multics is designed so that an unsophisticated user can remain oblivious to the interaction between file system modules and the remainder of his process. This is true because services of the file system will be invoked indirectly on behalf of the user by other modules such as by the Linker as a result of link faults. More advanced users may seek more direct contact with the file system by issuing file system commands (see BX.8) that result in calls to the Basic File System. These commands permit the user, for example, to obtain listings of his files, create or delete files including directories, modify branches by renaming them or altering access control entries, etc. As the user constructs subsystems having increased complexity, he is likely to need an increased understanding of the Multics file structure. In particular, he will need to see how modules of the file system maintain and control a dynamic interface between the file structure and his working process. In this chapter we hope to explain a number of the functions of the Basic File System with which the subsystem designer will be most concerned and over which he can exercise a degree of control.

We shall refer primarily to two modules of the Basic File System (BFS), Segment Management and Directory Control.

### Segment Management

(SMM) is responsible for properly interpreting the intent of the user's symbolic references to segments. Thus, it is the SMM that determines to which if any of the segments already "belonging" to the process does a given symbolic name refer. If to none, the SMM must then determine if any existing file in the hierarchy is to be associated with the user's symbolic name. If none, the SMM must then determine if a new file is to be created and placed in the hierarchy (on a temporary or permanent basis). Questions must be answered like: where in the directory structure should a newly created file be placed? Is the new file empty, or should it be a copy of an existing file? Finally, SMM must see to it that, however the file is identified, it is given the status of a segment in the executing process, that is the segment is made known via a segment number and appropriate access control is provided. Another way of saying this is that the target segment is put into the state where it may be properly written, read, or transferred to by the segment which referred to it.

The SMM modules are primarily managerial. Most of the actual work suggested in the foregoing paragraph is performed by other (conceivably more primitive or central) modules of the Basic File System, e.g., Segment Control, Directory Control, and Access Control. In this sense the SMM may be regarded as an interface with the core of the file system. It will not be practical to discuss the interface without some discussion of the core modules and of the data bases managed by the core modules.*

One of the data bases which will come under careful scrutiny now is the KST (Known Segment Table) to which we have made frequent, but glancing reference in preceding chapters. In simplest terms, the KST is a dynamically-maintained "registry" of the segments that are currently part of the process. Symbolic reference names and other identification are kept in the per-segment entries of the KST. This data also provides (ring) context information with which the SMM can determine the intended target when one segment makes symbolic reference to another segment that has previously been similarly referenced in the same ring.

## Directory Control (DC)

All user requests which deal with creation, deletion, alteration of files and/or their file descriptions ultimately result in invoking Directory Control to make the appropriate modification to the directory structure. All inquiries about the status or location of files and/or their descriptions also must ultimately invoke Directory Control, because only this module is permitted to read and alter the contents of the directory files.

Depending on the ring context, i.e., the ring in which a referencing procedure executes, the same symbolic reference name may be intended to refer to the same or to different segments (i.e., to different points) in the file system hierarchy.† Moreover, there may be several different reference names in use which are intended to

*Primary MSPM references for the material in this chapter are:

| | |
|---|---|
| BG.18 | Segment Management |
| BG. 3 | Segment Control |
| BG. 2 | Known Segment Table |
| BG. 7 | Directory Data Base |
| BG. 8 | Directory Control |
| BG. 9 | Access Control |

†Henceforth, we shall use the word "hierarchy" to mean "file system hierarchy."

refer to the same point in the hierarchy. These cases of multiplicity are sure to arise in subsystems involving groups of users, with several directories available as respositories for common routines. The SMM maintains control over these reference name-segment number pairings. It develops and reuses each name-number pair in its proper context.

## 6.2 DIRECTORY STRUCTURE

In Chapter 4 we began our discussion of the directory structure. Here, we shall review it and amplify it with the introduction to the as-yet-untreated subject of directory links. There are actually two types of entries which may be added to a directory — branches and links.

### 6.2.1 Branches

A branch, you recall, is a detailed description of a file. Among other things, a branch contains the physical locations in secondary storage of the records that comprise the file.[+] The file described in the branch may be another directory or, if it is not a directory, it can be thought of as a data or procedure segment. Any process that has access to a file is able to make it a segment in that process. In some cases detailed later it is a copy of the file, rather than the original which is made part of the process. Of course, when a copy is made it too must be identified as part of the file system. A branch which points to the file copy must be made and placed in an appropriate directory.[†]

[+] In point of fact, if the file extends over several records (1024 words each) the physical location takes the form of a file map, which is an ordered list of the addresses for the individual file records. A zero length file has an empty file map. As the file's maximum size is adjusted upward to range over one or more records, additional entries are made in the file map. So as not to take up unnecessary storage, a record of a file which has not yet been "written into," is regarded to have all its words in the zero state. Such zero records are not actually allocated secondary storage at the outset. Instead, the file map entry for this record is marked appropriately. Whenever a process makes reference to such a record (as a page of the corresponding segment), a block of all zeros would be written into core memory.

[†] If the copy is made for temporary use, i.e., only for the duration of the process which refers to it, the branch is placed in a directory that is associated with this process. It is appropriately called the process directory. Every process, while it exists in Multics, will have its own process directory.

At the time it is created, each branch has associated with it a unique identifier (in reality a generated (36-bit) serial number). In addition, each branch holds a list of one or more "entry names". These are the official nicknames or aliases by which we as users will normally refer to the segment in our source code. Multics is so designed that any entry name in the branch is a permissible reference name in our source code.

A principal purpose of the alias feature is to provide convenience to the programmer for designating symbolically distinct entry points in the same external segment. An appreciation of this concept is enhanced by explaining the following Multics convention. A call reference to an external segment of the form

"< procname>" in EPLBSA (or "procname" in EPL)

will be interpreted as the entrypoint whose address

"< procname> | [procname]" (or "procname$procname" in EPL).

The value of this convention becomes clear when we now consider a single procedure which has two or more aliases. Suppose we picture a library file whose branch has two aliases, "insert" and "delete". Then assuming insert and delete are declared external, an EPL call of the form

```
call insert (A, B);
```

will be interpreted by the compiler as syntactically identical with

```
call insert$insert (A, B);
```

while a call of the form

```
call delete (C, D);
```

would be interpreted the same as

```
call delete$delete (C, D);
```

and the respective targets will be

s# | insert

and

s# | delete

Here, we use the symbol "s#" to refer to the common segment number that would be returned by the Segment Management Module at the Linker's request. The offsets insert and delete are values which the Linker would obtain by inspecting the in-symbol table of the segment established for this file.

The converse situation should also be appreciated. Namely, if a file has only one entry name, say "delete", but the procedure it represents has two entry points, one named "delete" and the other named "insert", then to reach the second entry point, the user has no choice but to refer to it as delete$insert, i.e.,

call delete$insert (A, B).

Another purpose of the alias feature is to permit different programmers on the same project to refer to the same segment with different names, e.g., delete and remove. Rather than reassemble or recompile the code written by these programmers to force conformity among several programmers on the same project in the referencing of a given segment, it may sometimes be simpler to guarantee the appearance of both names in the file branch of such a segment, (and to let both names have the same offset value).*

Any programmer (in this instance, possibly the project manager), who has write access to the directory holding the particular branch, may add to it as many distinct entry names as he cares to. Of course, no two branches in any one directory are allowed to have a common entry name.† But, two or more directories may each have a branch with the same entry name. Figure 6-1a is an attempt to summarize the ideas concerning directory structure given in the foregoing discussions. To conserve space, circles represent directories and rectangles represent nondirectories in this figure (just the reverse of the technique used in Figure 4-1).

### 6.2.2 Links

A link is a special kind of named entry whose purpose is to point to another entry normally in some (any) other directory. Links permit a useful form of cross-referencing capability that can be superimposed over the basic tree structure formed by the branch type entries. Ordinarily, since files and their corresponding branches must be one-to-one, no two directories can directly share the same file. But, for example, with the use of link type entries that point to branches, a subsystem designer can effectively develop a directory so it appears to have files in it that, in fact, belong to other directories. Thus, in Figure 6-1b, directory q which "points" directly to only two files, named v and w, indirectly points to two others: h in n and c in s.

*This would probably mean revising the source code for this segment to make delete and remove synonyms (in EPLBSA) or dual labels (of the same EPL statement) and then recompiling.

†This restriction is guaranteed by the Basic File System.

Likewise, directory s indirectly points via a link to file m in the superior directory x. This capability for grouping under one directory a selected set of files which, via links are actually located in other directories, is a powerful "packaging" device in the design of subsystems. This packaging is especially effective when the subsystem is later embedded within other subsystems. We will return to this topic later on in this chapter.

Note, also, that if q is a directory belonging to user1 and n is a directory belonging to user2, then user1 may use the symbolic reference "r" while user2 may use the symbolic reference "h", each intending to access the same segment (<h>). Actual access to <h>, however, is another matter. User1's access to <h> is governed by the ACL information in the branch named "h" in <n>. The existence of a link to <h> in user1's directory has no bearing on user1's access privileges to <h>. Only users who have write access to <n> can accord user1 access privileges to <h>.

### 6.2.3 Path Names

There are several ways by which the system can refer unambiguously to a particular branch (or link) in the hierarchy. As for the subsystem writer, the primary way is by giving the path name for the entry.*

A path name, in the simplest sense, is a list of the node names from the root to the branch (or link) inclusive. Elements of the list are separated, not by commas as you might expect, but by the ">" character. Thus, the path name for the branch in Figure 6-1a named sub would be:

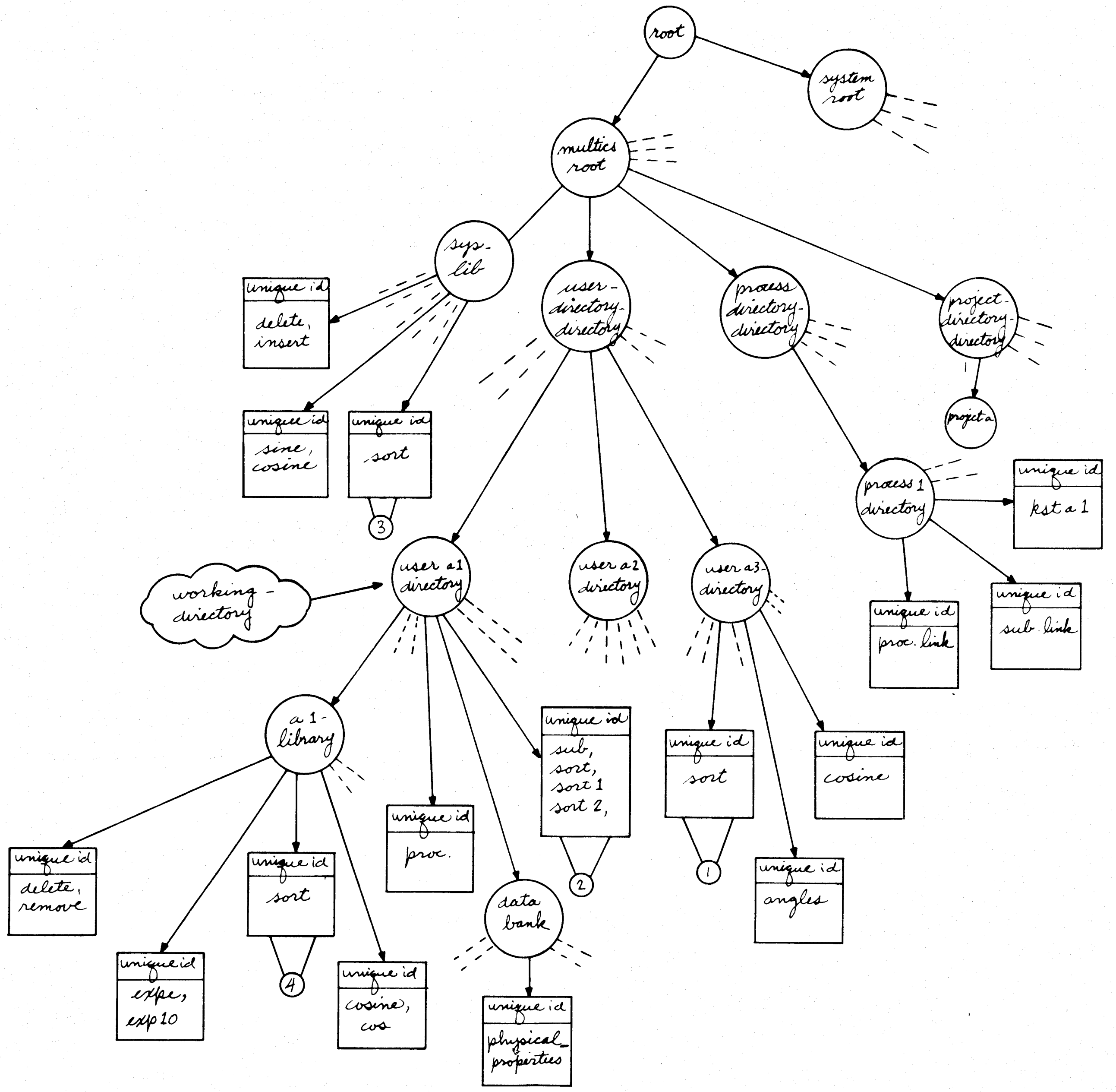
"root>multics_root>user_directory_directory>useral_directory>sub"

This part is normally never written

Since this same branch has the alternate entry name sort, an equally unique path name would be:

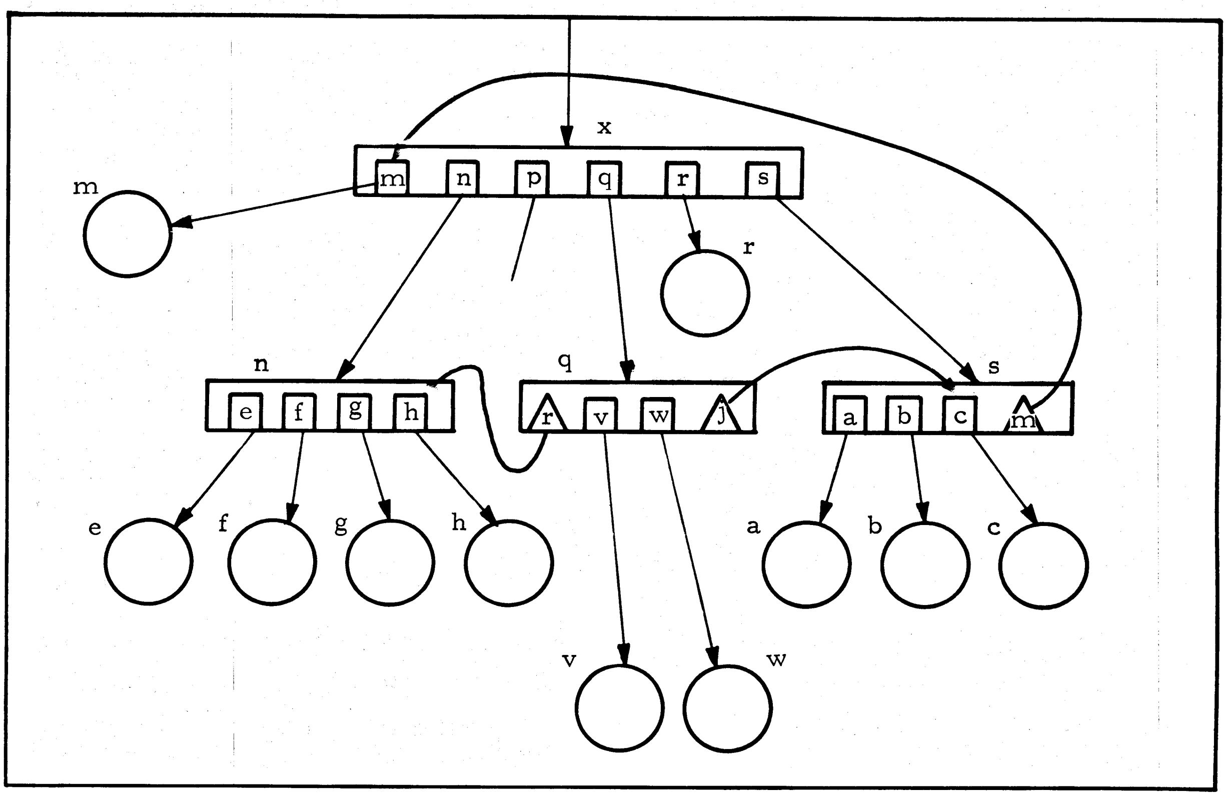
$$\text{"root>multics\_root}\underbrace{\underbrace{\text{>user\_directory\_directory>useral\_directory}}_{\text{directory path name}}\underbrace{\text{>sort"}}_{\text{entry name}}}_{\text{path name for the branch (or link)}}$$

*Another way, mentioned here only for the sake of completeness, is by unique id. Uniqueness is assured because the 36 bits include the date and time of day that the branch was created as well as the identification of the hardware system which created it. The ordinary user will not normally know the unique id for a segment he wants to reference. (See BG. 7 for more details.)



(a) Circles represent directory segments and squares represent non-directory segments. In reality, the names shown in the directory files (circles) and in the non-directory file (squares) are, in fact, stored in the branches to these files, i.e., in the immediately superior directory.

Figure 6-1a. Conceptual Model of the File System Tree Structure



□ branches

△ links

(b) Showing the cross-referencing that can be achieved with the use of links. Links may be independently named. Thus, in q, the link named j points to the branch whose name is c.

Figure 6-1b. Conceptual Model of the File System Tree Structure

Note that a path name for a branch can be thought of as having two main parts:[*] The path name for the directory, which points to the branch (*directory path name*), followed by the *entry name* for the branch.

Certain shorthand conventions are expected by the basic file system procedures. When you write a path name which emanates from the root node, you almost never write the name of the first two root directories of this path. Simply begin the path name with ">" as a shorthand for "root>multics_root>".[†] Thus any path name, e.g., ">a>b>c", which begins with ">" is considered to be an *absolute path name*.

### 6.2.3.1 *Relative Path Name*

An executing process has at all times associated with it a directory that is designated as the *working directory*.[‡] This is a directory that the process happens to be "currently using." It is merely a reference marker to a point in the hierarchy from which it becomes "convenient" to describe paths to other segments. Thus, tree paths to a particular node may be described relative to the working directory of a process. If a path name begins with some character *other than* ">", the given path is interpreted *relative to the working directory*. Use of this convention greatly shortens the length of most path names. We illustrate by referring again to Figure 6-1a. Assume that useral_directory is the working directory at some instant in time.

*Example 1*

The path name for proc is simply "proc"; for sub it is simply "sub". Thus, for branches (or links) in the working directory, the entry name and the path name are identical.

[*] A user can call for services of the basic file system directly via certain of its "primitives" in Segment and Directory Control, indirectly via such gates as the entry points in the SMM, or indirectly by using one of the many file system commands described in BX.8. In these calls (or commands), the target procedure in Segment or Directory Control either expects to have a path name passed to it as an argument, or expects to return a path name as an (output) argument. Typically, the caller is required to furnish the path name as a *pair* of arguments, i.e., *directory path name* and *entry name*. We will be speaking about some of these primitives in Section 6.4.

[†] It would only be used when making certain special calls directly to the Basic File System, enabling the caller to designate either "root>multics_root" or "root>system_root." The latter would only be recognized by a privileged callee. It is not necessary for a user to preappend "root>multics_root" because normally all path names are regarded as relative to either "root>multics_root" or to "root>system_root," whichever is stored in a key location within the pdf (process definitions segment), a special process data base available to the supervisor.

[‡] The system or the user is free to alter the designation of the working directory. BX.8.12 describes a series of commands, which allow the user to exercise direct control over the designation of the working directory. As control moves from user's procedures to supervisory modules, the working directory often changes temporarily, because supervisory modules (e.g., Directory Control) set the working directory and reset it as required.

Example 2

The path name for physical_properties (relative to the working directory is "data_bank>physical_properties."

Example 3

It is also possible to use the relative path name convention when referring to a branch that is not a descendant of the working directory. This is done with the aid of the character"< ". It is interpreted to mean parent of the working directory. Moreover, "<< " would mean parent of parent of the working directory, etc.

Thus, a suitable (relative) path name for <usera3_directory> is "<usera3_directory". This is a somewhat more attractive alternate to the (absolute) path name:

">user_directory_directory>usera3_directory"

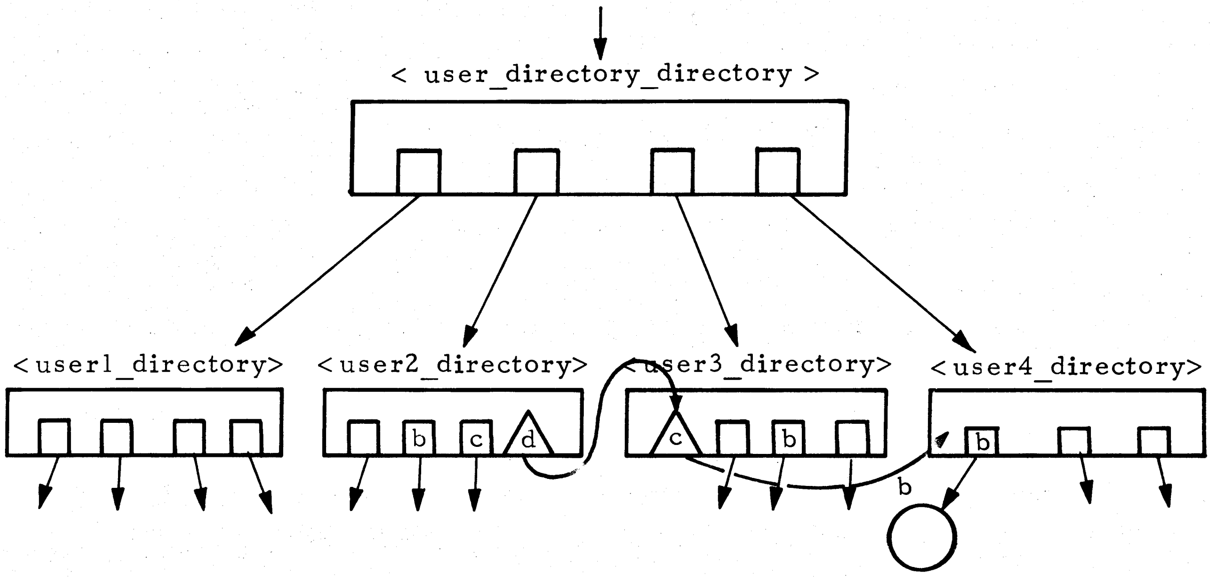
especially since users will not normally know the names of the parent directories.

Example 4

As another example of the same type, the (relative) path name for <angles> would be "<usera3_directory> angles".

### 6.2.3.2 Retrieving File Branch Information

When Directory Control is handed a path name for the purpose of retrieving corresponding file branch information, the desired directory entry is retrieved and its type (link or branch) determined. If it is a branch, the target has been reached; if it is a link, the path name found in the link is then employed for a repetition of the retrieval process. A chain of links eventually leading to a branch is also a possibility. Directory Control is coded to prevent the repeated retrieval process from getting out of hand, as could occur in the event the links loop back on themselves. Figure 6-2 shows a chain of two links leading to a file branch. The chain depicted in Figure 6-2 can conceivably arise in the following way. User4 grants permission to user3 to use the routine called "b". But, it is inconvenient for user3 to refer to the same procedure as "b" because he already has a routine in his directory called "b", so he chooses to refer to the user4 — procedure as "c". At some other time user3 persuades user2 to use the same routine, only user2 chooses to call it by still another name, "d", for similar reasons. When and if user2 makes a reference to "d", he may find he has no access to it, if user4 has made no provision to include user2 in the permission list for "b".



If user2 and user3 appear in the access control list for < b> in user4's user directory, then user2 may use "d" as a symbolic reference and user3 may use "c" as a symbolic reference to the file whose branch entry is named "b".

Figure 6-2. Chain of Links

### 6.2.4 Programming Without Path Names to Segments

If every programmer were forced to unambiguously designate every segment by its path name (absolute or relative), the job of the administrative modules would be made a great deal easier, but they would have few, if any, customers! A major design objective of Multics is to shift the burden of unambiguous segment designation, or as much of the burden as possible, from the user to the system. Certainly, for example, the ordinary PL/I or FORTRAN programmer should be entirely freed of this burden.

Our purpose now is to show in a general way how this is done and greater details will be provided in later sections.

In ordinary usage, the reference name given for a segment is the name given to the corresponding file's directory entry (normally a branch). This reference name can be thought of as the path name for the file stripped of its front end (i.e., stripped of its directory path name). It is this omission which, while making it easier for the users, complicates the problem for the SMM when it is asked to obtain a segment pointer for the symbolically referenced segment.

We illustrate this problem by again referring to Figure 6-1a. Suppose the procedure named proc is executing in a process which we shall call process1. Now, let < proc>, in particular, be the segment whose branch is in <useral_directory>. Further, let us suppose < proc> now incurs a link fault to someplace in < sort>. The Fault Interceptor passes the baton to the Linker which, after extracting the string "sort" from < proc>'s outsymbol table, now calls the SMM, asking it to return a segment pointer to < sort>, i.e., an its pair of the form

| | | |
|---|---|---|
| sort # | 0 | its |
| 0 | 0 | 0 |

It is possible that the SMM already knows the segment number for this segment by virtue of having previously helped the Linker on an identical problem, e.g., on a link-fault to the same segment. There is no problem in this case. This is because the SMM oversees the recording of all such symbolic reference name-segment number pairings in the process' KST (Known Segment Table). Hence, when requested to do so, the SMM will cause the KST to be searched for the return of the matching segment number.

If, however, there is no record of the reference name "sort" in the KST, then the SMM has the problem of deciding which of the many possible segments named "sort" is the one desired by the faulting procedure. Looking over Figure 6-1a, we see four branches, each having an entry name "sort". Some are more "likely" than others, but, in principle, any one of these may be the intended segment.

Most likely, however, the one marked ① is not intended. But, any one of the other three, i.e., ② , ③ and ④ might very well be the one the user had in mind.

### 6.2.5 Standard Search Rules

Multics has adopted a standard approach (actually a heuristic) for resolving this ambiguity, consisting of a sequence of trial assumptions. The first trial assumption is that the intended target name is a file having a branch in the same directory as the one holding the branch for the faulting procedure segment. That is, the first directory to be considered is the so-called caller directory abbreviated as "cdir". In our example, cdir for < proc>, the faulting segment, is "useral_directory". Hence, the branch marked ② in Figure 6-1a would be selected. In general, however, if no entry can be found in cdir having a name that matches "sort", the SMM falls back on a secondary search strategy. This is to search a (system-prescribed) ordered list of directories. First on this list is the user's current working directory dubbed "wdir". (Frequently cdir and wdir will be the same.) Following this is a list of system library directories, and finally the process directory. The first match found is then considered to identify the intended target. If no match is found in any of the prescribed directories in the "search list", failure is then conceded by the SMM. It reports an error code back to its caller (normally the Linker), which in turn must act appropriately.

To summarize, the Standard Search List is given in Table 6-1.

TABLE 6-1

Standard Search List

| | | Directory | Name (or metaname) |
|---|---|---|---|
| Primary Strategy | 1. | The directory holding the branch to the faulting procedure | (cdir) |
| Secondary Strategy | 2. | The user's working directory | (wdir) |
| | 3. | The Multics Command and Systems Library | Sys_lib |
| | 4. | A list of up to five additional special libraries which may eventually be placed in the hierarchy | Sys_lib_1 |
| | 5. | | Sys_lib_2 |
| | 6. | | Sys_lib_3 |
| | 7. | | Sys_lib_4 |
| | 8. | | Sys_lib_5 |
| | 9. | The user's process directory | (pdir) |

### 6.2.6 Permission to Search

The scan of each directory in SMM's search list is performed by Directory Control. Each search of a directory is done on behalf of the faulting segment. Consequently, Directory Control, a ring-0 module, first ascertains the faulting segment's right-of-search. In essence, the rules are as follows. Suppose the faulting segment is < a> and it needs to know if "b" is an entry name in <directory1>. Although Directory Control does the "looking," it first checks that the user has search privileges in <directory1>. This check amounts to determining if the user_id (for the process which is calling Directory Control) appears in the access control list for the branch describing <directory1> and, moreover, that the E (execute) attribute is ON in the effective mode of this branch.* For a refresher on what the user_id is, review Section 4.2.2 of this Guide.

If any directory on the search list fails to meet these criteria, it is skipped and the next directory on the list is searched.

As another example, if < proc> took a link-fault using the reference cosine, † a path name to the branch named "cosine", found in Sys_lib, would be returned. If, prior to taking this link-fault the user had executed the set_wdir† command, designating the path name: "<useral_directory>al_library" as the working directory, then his own library cosine routine would be selected instead of the Multics library cosine routine.

Finally, we note that in the hypothetical situation given in Figure 6-1a if useral's < proc> takes a link-fault to sort while the working directory is set at useral_directory, it would be impossible for the SMM to return a pointer to the sort procedure in al_library, i.e., to ④ unless the user makes some deliberate effort to achieve this objective. This is because useral already has a branch named sort (one of four aliases) in the same directory that holds < proc>, the caller.

In order to give < proc> the opportunity to reference useral's library routine called sort, one of two approaches must be taken. a) Prior to the call to sort, make an explicit request of the SMM to "initiate" the desired file as a segment which, in this process, is hereafter to be referred to as sort. To initiate a segment in this

*The principal MSPM reference is BG.8.04.

†Henceforth, reference names will be italicized (underscored) in this text so as to avoid using quotation marks.

† See BX.8 for a description of this command.

way a programmer must be able to furnish the SMM with the path name of the desired file. Note that the same problem can recur again during execution of another, similar process. The explicit call to the SMM for initiating the desired sort file does not alter the basic directory structure or search strategy, so it only provides a per-process "remedy". We shall defer additional discussion of this type of expedient until Section 6.4. b) Make some change to the directory structure itself anytime prior to the implicit invocation of the SMM. One possible change which might be made is as follows: First, delete the alias sort from the list of four names given to the branch marked ② . (The delname command can be used for this purpose, as described in BX.8.06.) Second, add a link to useral_directory which points to the branch named sort in al_library, as sketched in Figure 6-3. (The link command can be used for this purpose, as described in BX.8.04.) An alternative to this second step would be to have < proc> execute a call to the system library routine, change_wdir to set the working directory to "al_library" immediately prior to the call on sort. The call would, in this instance, be:

```
call change_wdir ("al_library").
```

After returning from the desired sort routine it would probably be advisable to reset the working directory by issuing the call:

```
call change_wdir ("useral_directory").
```

The foregoing actions exemplify the type of control a user can exercise on the directory structure that is subsidiary to his user directory. In later sections, especially Section 6.5, we shall consider the case where the same reference name may be used (in the same process) to mean different files in the hierarchy. This conflict-of-names problem and ways to solve it or avoid it is one with which subsystem designers will frequently be confronted.

### 6.2.7 The Case of a Search Failure

If after considering every directory on the search list no entry is found, this failure will be considered to be an error and the error code is returned by the SMM to the Linker, and thence to the Fault Interceptor which signals a link-fault condition in the ring of the faulting segment. If the user has not provided his own handler, a system-supplied default handler will be invoked. It will print a canned message and will then call a system procedure to terminate the process. In this way the nature of the error is reflected back to the user.

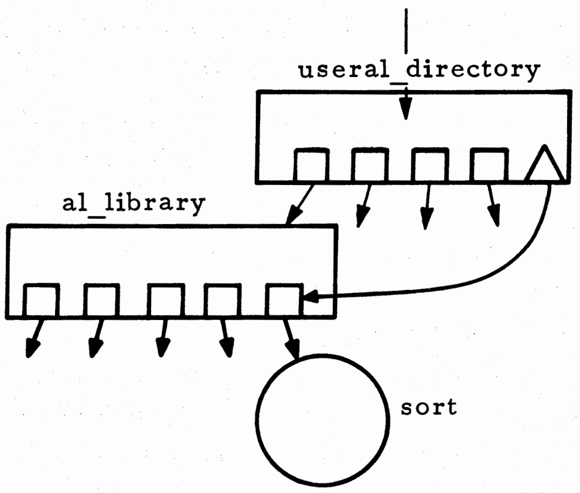


Figure 6-3. Linking to Branch Named Sort

6.2.8 Access Failures ("The operation was successful but the patient died".)

Even if the search is successful, Access Control which is called by Directory Control after the branch has been found, can still reject the request on grounds of access violation. Access Control, as described in Chapter 4, will check to see if the user's user_id appears on an ACL (Access Control List) entry in the branch. Failure to find the user_id (or a class name which includes user_id as a member) in any ACL entry of the branch is, in fact, only a partial indication that the creator of the target file wishes access to be denied to this user. Each directory is actually provided with another access control list. It is called the Common Access Control List (CACL). A user_id that appears in a CACL entry of a directory is accorded blanket access to the files for all branches in that directory. Access Control checks the CACL* after failing to find the user_id in the ACL of the target branch. A failure to find a listing on the CACL seals the verdict. Access is denied and the rejection

*In Chapter 4, we deliberately omitted a discussion of the CACL when we described the structure of a directory. The reader can see these details by inspecting BG.7.

notice is passed backward via the SMM to the Linker, etc. The nature of the error is reflected back to the user in the same way as we described for the case of a search failure.

## 6.3 INITIATING A SEGMENT

Assuming the search module has been successful, the SMM now has a complete path name for the desired file. Its next job is to initiate this file as a segment to the current process. From the viewpoint of the user, initiating a segment simply means establishing a name-number pair for use in handling this and all future symbolic reference (including link-faults) that are directed toward the same segment.

The information associated with each initiated segment is registered as a new entry in the Known Segment Table. Placing an entry in this table for a new segment is referred to as making a segment "known" to a process. Placing the $i^{th}$ entry in this table is tantamount to assigning i as the segment number of this segment. Each entry is threaded back to the entry that corresponds to the immediately superior directory file. In this way, KST entries form a tree structure. In essence, the KST tree is a subtree of the entire file system hierarchy and characterizes the current state of the executing process.

Table 6-2 lists the items in each KST entry.* Item 3 serves as the backward thread referred to in the preceding paragraph. The very first item is a dynamically maintained list of reference names by which each segment is known in the process. More will be said about this item momentarily. Items 6, 7, 8, 9, and 10 represent information copied from the corresponding branch at the time the segment is initiated.† The effective mode and ring brackets (items 7 and 8) are used by Segment Control to construct (and possibly later to revise) SDW's (segment descriptor words). Items 2, 4, and 5 will be disregarded in the present discussion.

Only the Basic File System is privileged to use or modify KST data. This is because the data kept here is the basis for policing the process' use of each segment, and for this reason must remain compatible with the latest access control information given in the branch for this segment.

*A complete description of the KST and its storage structure is given in BG.1.

†Items 7, 8 and 10 are updated from time to time as a result of subsequent changes made in the corresponding branch.

TABLE 6-2

List of Items in a KST Entry

| Item No. | Importance level of item for discussion in this chapter (1 is most important) | Item Description | Applicability Check on the type of Segment | |
|---|---|---|---|---|
| | | | Directory Segment | Non-directory Segment |
| 1 | 1 | List of names* | ✓ | ✓ |
| 2 | nil | Number of currently known segments in this process for which this directory is the parent directory | ✓ | ignore |
| 3 | 1 | Pointer to the branch for this segment. (Includes segment number for this segment's parent directory | ✓ | ✓ |
| 4 & 5 | nil | Transparent usage and transparent modification switches | ✓ | ✓ |
| 6 | 4 | Directory segment switch (ON if a directory) | ✓ | ✓ |
| 7 | 1 | Effective mode (R, E, W, A) | ✓ | ✓ |
| 8 | 1 | Ring brackets (r1, r2, r3) | ✓ | ✓ |
| 9 | 3 | Unique identifier (36 bits) | ✓ | ✓ |
| 10 | 2 | Date and time the branch for this segment was last modified | ✓ | ✓ |

*If this entry is for a directory, the list begins with the absolute path name for the directory. The remaining names constitute a cumulative list of symbolic reference names by which the process currently knows this segment.

### 6.3.1 Listing Symbolic Reference Names in the KST

In a single process a target segment may be referred to by one of several different names. The first of these names that is actually used as a symbolic reference will trigger the initiation of the segment (i.e., the construction of a KST entry). The segment will then initially be known by this first symbolic reference name. Later, if a second reference name is used and if the SMM determines that this too is a valid alias for the same segment, this alias is also entered into the KST entry (as an appendage to the list in Item 1 of the entry).

Any of the following are valid reference names for a file.

1. Entry names in the file branch for this segment.
2. Entry names in a link to the file branch for this segment.
3. Any name that the user may wish to declare (by an explicit call to the SMM) to be an alias for the segment. Such declared aliases are temporary, i.e., for the life of the process. Since the KST vanishes when the process is destroyed, so does the temporary alias. Section 6.4 mentions how these alias declarations are made.

#### 6.3.1.1 Ring Context Prefixes

Each such name that is stored in the KST entry has prefixed to it an important bit of context information, namely the validation level.* This number is the ring in which the referencing procedure is executing when it makes the symbolic reference. The KST form is:

$$\text{"}\underbrace{\text{nn}}_{\text{validation level}}\text{\_refname"}$$

If the SMM is subsequently asked for a segment pointer to the same "reference", mere discovery of this name in a KST entry will not be sufficient; the prefixed validation level must also be matched against the current validation level. Only then will the SMM deduce that the index of the KST entry where the match was found is the appropriate segment number for use in constructing the requested segment pointer.

By associating the validation level, nn, with each recorded reference name in the KST, the SMM is able to offer the user an extra degree of control over the mapping of names to their intended target segments. A user may, if he chooses, use the same reference name to mean two or more different segments, each target being determined in the context of the ring in which the reference to it has been made.

*For a refresher on validation levels, see Section 4.3.4.

### 6.3.1.2 Examples

A somewhat elaborate and admittedly contrived series of examples is given here to illustrate the points we have just made. We assume that the file system hierarchy includes the files and file branches as shown in Figure 6-4. We further suppose that in usera5's process a sequence of symbolic references are made in the order shown in Table 6-3.

Each line in the table gives values for the pertinent state variables (columns 2, 3, and 4) that were extant at the time the symbolic reference was made. It also shows the identified target segment (column 6) and the name if any that has been added to the corresponding KST entry (column 7). The response of the SMM (columns 5-7) for a given line in the Table is clearly dependent on the process' history. For our purposes, the history begins at line one.

The table must be read one row at a time. As you read a row you are expected to be consulting Figure 6-4. The first 11 lines show the process executing a chain of calls:

$$p \longrightarrow q \longrightarrow r \longrightarrow s \longrightarrow t$$

The next paragraphs amount to a walk through the first few lines of the table. Depending on your interest, you are invited to finish the walk through the first 11 lines as one exercise and if the spirit really moves you, to complete the remaining lines as an additional exercise.

#### Walk Through Lines 1 through 6 of Table 6-3

#### Line 1

Procedure <p> executing in ring 32, using <usera5_dir> as its working directory, makes a reference to a segment using the name a. Since this is presumably the first time the reference name a has been used in this process, there will be no KSTentry having a reference name of the desired form (32_a), so a search of the hierarchy is begun, beginning with the caller's directory. The caller directory is <usera5_dir> and a search of this directory finds a branch (to file $\boxed{5}$ ) with an entry name a. The KST is re-searched to see if there already exists an entry whose unique id is $\boxed{5}$.* This could be the case if the file had previously been initiated under a different alias. We assume here that this possibility in fact did not occur. Hence, a new KST entry is created and the entry name 32_a is added to its (empty) list of reference names.

*It should be clear that our use of a number inside a square to represent a unique identifier (actually 36 bits), is merely a graphical convenience.

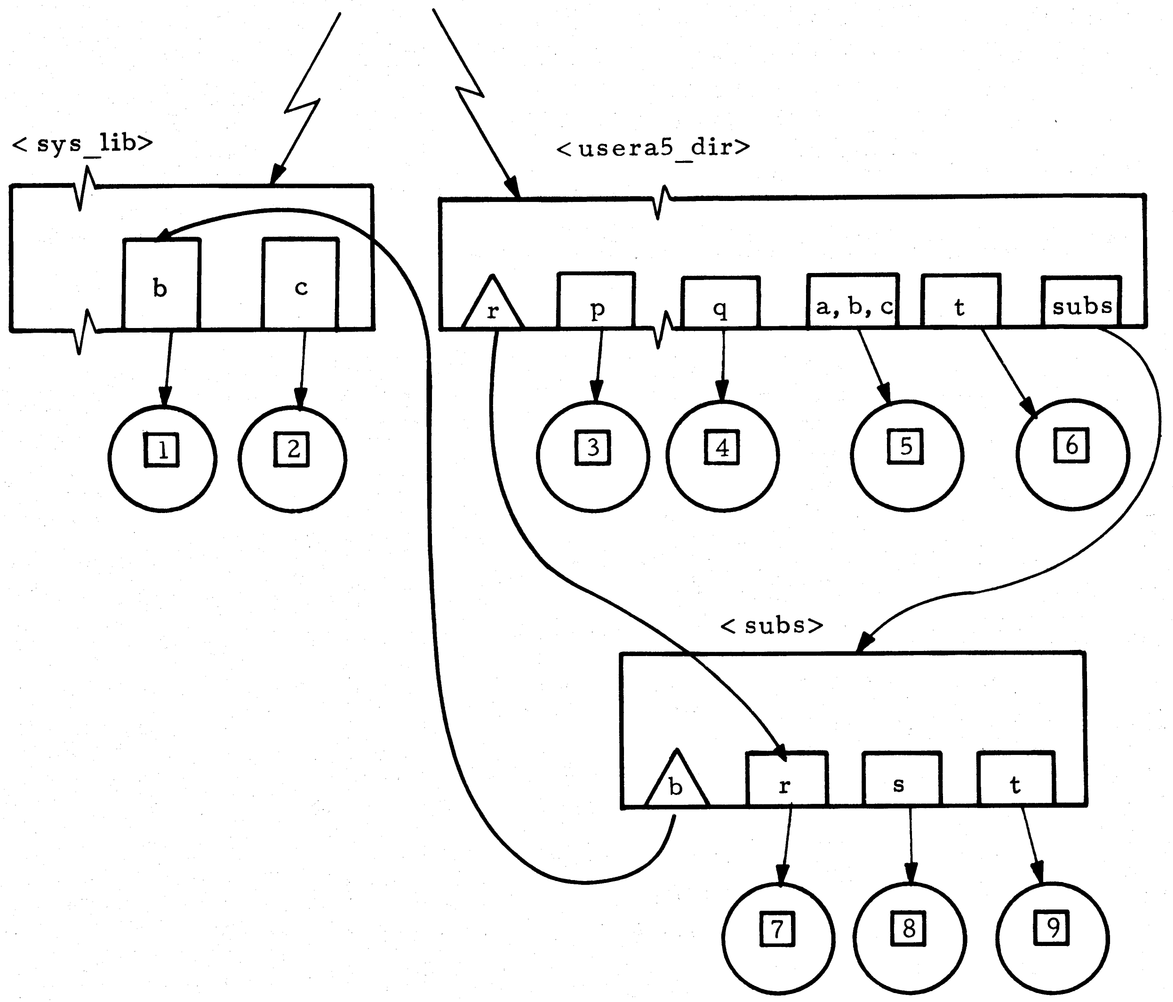


Figure 6-4. The File System Hierarchy

TABLE 6-3

Buildup of Entry Names in the KST

| | Line No. | ① | State Variables | | | Response by the SMM | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | ② | ③ | ④ | 5a | 5b | 6a | 6b | 7a | 7b |
| | | Name of referencing procedure | Ring of execution | Current working directory | Symbolic reference name | Directory search required (yes or no) | Name of Directory in which matching entry is found | Unique designation (schematic) of the target file and its corr. segment | New KST entry (yes or no) | New KST entry name | Form of new KST entry name |
| <p> | 1 | <p> | 32 | <usera5_dir> | a | yes | <usera5_dir> | $\boxed{5}$ | yes | yes | 32_a |
| | 2 | <p> | 32 | <usera5_dir> | q | yes | <usera5_dir> | $\boxed{4}$ | yes | yes | 32_q |
| <q> | 3 | <q> | 32 | <usera5_dir> | b | yes | <usera5_dir> | $\boxed{5}$ | no | yes | 32_b |
| | 4 | <q> | 32 | <usera5_dir> | r | yes | <subs> | $\boxed{7}$ | yes | yes | 32_r |
| <r> | 5 | <r> | 33 | <usera5_dir> | b | yes | <sys_lib> | $\boxed{1}$ | no | yes | 33_b |
| | 6 | <r> | 33 | <usera5_dir> | c | yes | <usera5_dir> | $\boxed{5}$ | no | yes | 33_c |
| | 7 | <r> | 33 | <usera5_dir> | s | yes | <subs> | $\boxed{8}$ | yes | yes | 33_s |
| <s> | 8 | <s> | 33 | <usera5_dir> | b | no | - | $\boxed{1}$ | no | no | - |
| | 9 | <s> | 33 | <usera5_dir> | c | no | - | $\boxed{5}$ | no | no | - |
| | 10 | <s> | 33 | <usera5_dir> | t | yes | <subs> | $\boxed{9}$ | yes | yes | 33_t |
| <t> | 11 | <t> | 33 | <usera5_dir> | c | yes | <usera5_dir> | $\boxed{5}$ | no | yes | 32_c |
| | 12* | <q> | 32 | <usera5_dir> | a | | | | | | |
| | 13 | <q> | 32 | | t | | | | | | |
| | 14 | <t> | 32 | | b | | | | | | |
| | 15 | <t> | 32 | | r | | | | | | |

*It is presumed that prior to reaching the occasion of line 12 normal return has been executed from <t> to < s> to <r> to < q> .

### Line 2

Procedure <p> (still executing in ring 32) calls <q>, thereby making the symbolic reference q. Since (we assume that) no entry in the KST has the reference name 32_q, a search of the hierarchy yields the fact that file $\boxed{4}$ in <usera5_dir> fulfills our search requirements. Assuming a second search of the KST shows that no entry now has the unique id equal to $\boxed{4}$, the appropriate KST entry is then formed and the name 32_q is added to its list of reference names.

### Line 3

Procedure <q> executing in ring 32 makes a reference to a segment using the name b. Assuming the search of the KST fails to find an entry name that matches 32_b, a search of the hierarchy will result in again finding file $\boxed{5}$. A re-search of the KST now shows there already is an entry for a segment whose unique id is $\boxed{5}$. So, no new KST entry is needed. It is merely necessary to add 32_b to the list of reference names in the existing entry.

### Line 4

Procedure <q> calls <r> with the symbolic reference r. A search of the hierarchy is again assumed necessary. This time Directory Control will discover that a link in <usera5_dir> (not a branch) has the matching entry name. The path name found in this link points to a branch in <subs>, thus identifying file $\boxed{7}$ as the target. Again, we assume a new KST entry for this segment must be made on grounds that there is no existing KST entry having $\boxed{7}$ as its unique id.

### Line 5

Since <r> executes in ring 33, when <r> makes symbolic reference to b, a match must be found with 33_b in a KST entry. No such match will be found in this case so again the hierarchy is searched. But now, the caller directory is <subs>. This is the first directory in the search list now. The link named b is found in <subs> leading Directory Control to come up with file $\boxed{1}$ in <sys_lib> as the intended target. A new KST entry is formed, initiating file $\boxed{1}$ as a segment, and making it known by the name 33_b.

### Line 6

Procedure <r> refers to c. There is no branch nor link named c in the caller directory <subs>. The standard search rules next dictate a search in the working directory, <usera5_dir> . The search succeeds in locating the branch for file $\boxed{5}$ , one of whose aliases is c. The reference name 33_c is now added to the list of names by which the segment for file $\boxed{5}$ is now known.

In case the subsystem actually intended that the target be file $\boxed{2}$ in < sys_lib> , it would be necessary to have previously established a link named c in < subs> pointing to the branch named c in < sys_lib> .

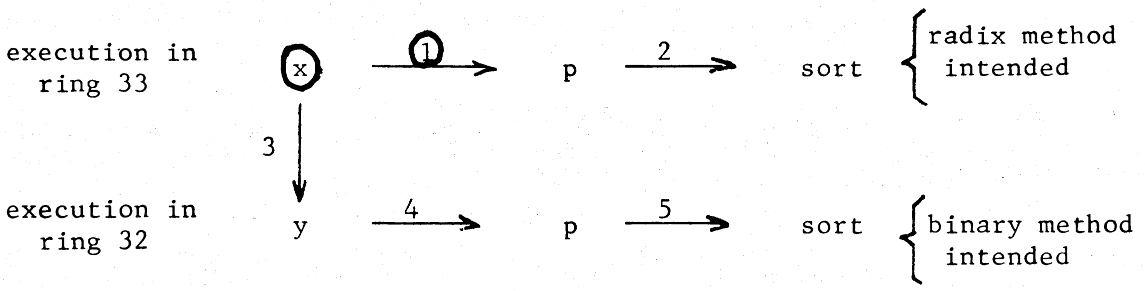
### 6.3.2 More on Ring Context Implication

If a procedure < p> has an access bracket of two or more rings, it is possible (and perhaps occasionally desirable) for the meaning of < p>'s symbolic references to depend on the particular ring within the access bracket in which < p> happens to be executing. To be more specific, Figure 6-5 shows a sequence of symbolic references which will help to explain the point we have just made. In this situation the service procedure named p can execute in ring 32 or in ring 33. Rules for determining in which ring of its access bracket a procedure will execute were developed in Section 4.2.2.1. Applying these rules we see that if called from procedure < x> whose ring brackets are (33, 33, 33), then < p> will execute in ring 33. If called, say, from a procedure < y> whose ring brackets are (32, 32, 32), then < p> will execute in ring 32.

Figure 6-5 suggests that while < p> executes in ring 33 and references sort, the intended target segment could very well be one routine, say, a radix sorting procedure; whereas an entirely different target is intended, say a binary sorting routine, when < p> executes in ring 32. Moreover, if the thread of control shifts back and forth between ring 32 and ring 33, the meaning of sort, as used in repeated calls from < p> could alternate!

One way this alternation of meaning for a symbolic reference could be explained is by picturing that there is a physically different copy of the link pointer used to address the target. Each of the two link pointers would be snapped by the Linker, but to distinct targets. As a matter of fact, this is precisely how this dual intent occurs in Multics. The two different link pointers reside in different copies of linkage segments for < p>. One linkage segment is used when < p> executes in ring 33 and the other is used when < p> executes in ring 32.

We shall now take a more complete look at the idea of multiple copies of linkage segments. Having done this, we will consider how the situation described in Figure 6-5 is appropriately reflected in the KST.



This sequence begins with execution in a segment known by the reference name x. Access bracket for p is (32, 33). When p, referenced by x, refers to sort, a different target is intended than when p, referenced by y, refers to sort. This dual intent is achieved by providing two copies of p's linkage segment, one for use when executing in ring 32 and one for use when executing in ring 33.

Figure 6-5. A Sequence of Five Calls

Ring protection considerations dictate that for each ring in which < p> executes a separate copy of the linkage section must be used. Here is why: While < p> is executing in ring 32, the corresponding linkage data must be protected from procedures executing in higher-numbered rings. Hence, the ring brackets for < p>'s linkage section must be of the form (i, j, 32), where $i \leq j \leq 32$. Now, we can employ a similar argument when < p> is executing in ring 33. Granted, that < p> must have access to its linkage data, but how can it have access to the particular linkage section made earlier whose ring bracket is (i, j, 32) ? Clearly, a new copy of the linkage section must be formed whose ring brackets are of the form (j, k, 33), where $j \leq k \leq 33$. Extending this argument to the general case, we see that a procedure < p> whose access brackets are ($\ell$, m) may require (as necessary) a separate copy of its linkage section for each of the rings $\ell$, $\ell$+1, ..., m-1, m in which < p> actually executes.*

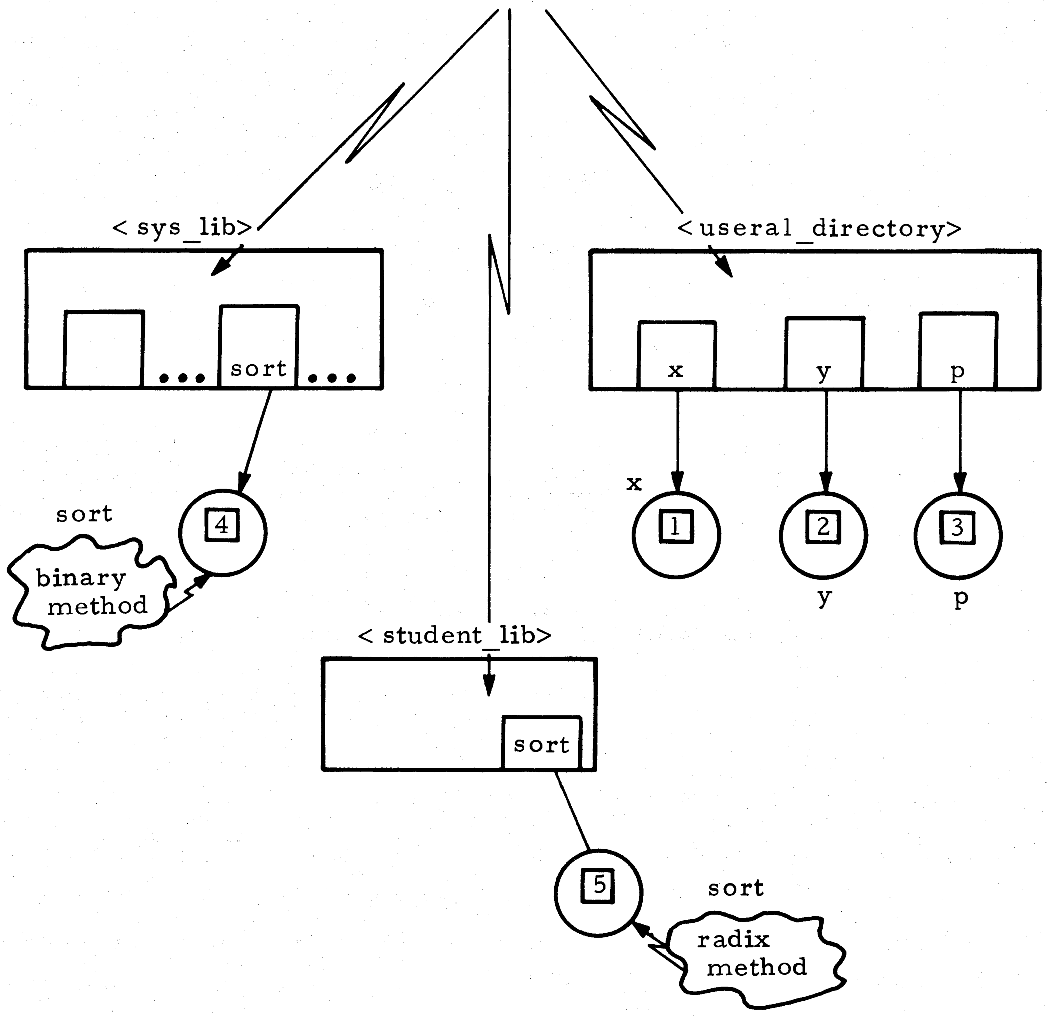
### A Matter of Terminology Regarding Linkage Sections.

> In the foregoing paragraph you may have noticed we used the phrase linkage section rather than linkage segments. It would be very costly if the system let each new linkage copy stand as a separate segment. Some of the consequent costs are: longer descriptor segments, longer KST's (and other such ring-0 segments whose entries are on a per-segment basis), wasted space in individual linkage segments and more segments which can be missing from memory when they are needed. To avoid these expenses, the Linker whenever possible will place each created copy of a linkage block onto a special, one-per-ring segment called a combined linkage † segment. This segment contains linkage blocks or sections for other segments which have also been referenced in this ring; call it r. The combined linkage segment for this ring has the ring bracket (r, r, r). It is actually of no great importance to the subsystems writer whether a linkage section "stands alone" or is combined with those of other segments. The important net effect relative to this discussion is the same; namely: multiple copies of linkage sections are made for a segment when it is referenced in different rings. It is, however, far simpler for our discussion to picture each of these linkage copies standing as separate segments. So in all subsequent discussions we will revert to the terminology of linkage segments rather than linkage sections.

We are now ready to view the action taken by the SMM in handling the five symbolic references depicted in Figure 6-5. This is done with the aid of Figure 6-6 and Table 6-4. The former shows the kind of directory structure which would be needed to support the "ambivalence" of sort in the Figure 6-5 example. What appears to be required is that branches for neither of the two targets called sort may appear in the same directory as the branch for < p>, the calling procedure. Line numbers in Table 6-4 correspond to the 5 numbered symbolic references of Figure 6-5. It is necessary that the working directory be adjusted at least once during the sequence of references. Two changes are assumed in the trace depicted in the table. Wdir is assumed to be set to < student_lib> prior to the ring-33 call on < p>. Wdir is set again, this time to < sys_lib>, prior to the ring-32 call on < p>.

*Certainly these copies need not be made in advance. The Linker, in fact, orders these copies to be made as it handles link faults to < p> from each new ring in the range $\ell$ through m.

†The primary reference is MSPM BD.7.05.



This structure is consistent with the call sequence in Figure 6-5.

Figure 6-6. A Possible Directory Structure

# TABLE 6-4

Build-up of KST Entries for Case Shown in Figures 6-5 and 6 -6.

| | | State Variables | | | Response by the SMM | | | | Remarks |
|---|---|---|---|---|---|---|---|---|---|
| Line No. | ① Name of referencing procedure | ② Ring of execution | ③ Current working directory | ④ Symbolic reference name | 6a Unique id of target | 6b New KST entry required | 7a New KST entry name | 7b Form of new KST entry name | |
| 1 | <x> | 33 | <student_lib> | p | $\boxed{3}$ | yes | yes | 33_p | First copy of <p. link>is made. |
| 2 | <p> | 33 | <student_lib> | sort | $\boxed{5}$ | yes | yes | 33_sort | Radix sort is selected. |
| 3 | <x> | 33 | <student_lib> | y | $\boxed{2}$ | yes | yes | 32_y | Wall crossing fault (inward). |
| 4 | <y> | 32 | <sys_lib> | p | $\boxed{3}$ | no | yes | 32_p | <y>sets working direction to <sys_lib> before making call on p. Second copy of <p link> is made. |
| 5 | <p> | 32 | <sys_lib> | sort | $\boxed{4}$ | yes | yes | 32_sort | Binary sort is selected. |

## 6.4 EXPLICIT CALLS TO THE SMM

Thus far we have been viewing the SMM as a module that is normally called by the Linker in search of a segment pointer for use in snapping a link to a target segment. Now that we understand the task involved in initiating a segment, we can consider reasons why a user may wish to cause the initiating or terminating of a segment in a more explicit manner, i.e., by calling the SMM directly.*

One important reason is so that a user process can be allowed to declare that an arbitrary alias is to be meaningful as a reference name for a particular segment of the process. The declared pairing of reference name and segment, which is defined by giving its full pathname, can then remain in effect throughout the life of the process. Temporary aliases are not made part of the directory structure since they are stored only in the KST entry for the stated segment. Subsequent to the declaration that "initiates" the alias, (in actuality a call to the SMM entry point, hcs$initiate), link fault references can be made to the declared alias just as if it were a valid entry name in the directory hierarchy.

It may also be necessary occasionally to ask for the initiation of a segment in order to learn information <u>about that segment,</u> possibly in anticipation of referencing the segment itself at a later time.

Table 6-5 contains a selected list of tasks which a user's process may need to have completed. These tasks can in each instance be accomplished by the appropriate call or sequence of calls to SMM entry points.

Most of the listed calls to the SMM are relatively safe for a subsystems writer to make -- with the exception of a call to terminate a segment. The SMM is nice enough to return a value for an error code argument that indicates the success or failure of its mission. (The reader can consult BG.18.01 for details on the calling sequences, if he chooses.) Thus, an attempt to initiate a segment from ring 32 with the reference name <u>zilch</u> will fail if there already exists a KST entry with the name "32_zilch" and the returned error code will so indicate.

To initiate a segment, a user must supply the SMM with a path name. To terminate a previously initiated segment, a user must supply a segment number. Herein lies a real danger. This segment number is normally freed for a reassignment (by analogy to a reuse of a telephone number) the next time the process initiates another segment. When a segment number n is terminated, KST entry numbered n is deleted and the SDW numbered n is set to zero to induce a segment fault the next time access is attempted through it. As long as this number remains "deactivated" in this way, links that have been previously

*A complete treatment will be found in the BG.18 sections of the MSPM.

TABLE 6-5

Tasks that a User's Process May Need Completed

| Task | EPL Names for the Appropriate Entry Points in the SMM that are to be Used |
|---|---|
| 1. Given a valid path name for an existing file, initiate a segment and obtain a pointer for it. Optionally, supply a reference name as well, so that it will thereafter be associated with the segment being initiated. Optionally, supply a reference which is to be regarded as a copy of the one identified by the given path name. | hcs_$initiate |
| 2. Given a particular reference name find a file in the search list of directories for this process. | hcs_$get_segment (if the path name for the desired target can be found, the corresponding file will be initiated as a segment by an appropriate call to hcs_$initiate). |
| 3. Find the segment number for a segment, given its reference name. | hcs_$get_seg_ptr |
| 4. Find out the path name of a segment, given its segment number. | hcs$get_path name (if not previously initiated a returned an error code tells you so.) |
| 5. Find out the effective mode and ring brackets for a segment given its segment number. | hcs_$get_seg_status (if not previously known, first call hcs_$ initiate). |
| 6. Find out any or all of the reference names by which a given initiated segment is currently known. | hcs_$get_name (use this entry point one or more times as needed). |
| 7. Create an empty segment (file) by a given name and in a designated directory having certain designated attributes, e.g., size, effective mode. | hcs_$make_seg |
| 8. Terminate a segment identified by a particular segment number. (Terminating a segment is the inverse of initiating it.) | hcs_$terminate |

snapped to segment n will induce segment faults and the user will be alerted via error messages if such unintended events occur. However, if prior to executing these snapped links, the user's process initiates another segment which is assigned segment number n, chaos can surely occur. Any unintended address formation which re-employs the snapped links to the former segments will go undetected. Errors and possibly faults of unpredictable nature (and of hard-to-recognize origin) are likely to occur shortly thereafter.

As you can see, an ordinary call to terminate a segment involves some risk. A user should be certain that his process will never re-employ previously snapped links while the target segment is terminated. In terminating a segment the user may optionally specify that its segment number be held in reserve, i.e., not returned to the "pool" of available segment numbers. Exercising the option eliminates the risks described above and also offers other interesting possibilities which may be of positive use to the subsystem writer. To appreciate the latter possibilities, one must also be aware of a companion option that may be exercised in the call to hcs_$initiate. This option permits the user to specify a segment number to be assigned (if available) for the segment being initiated. Employing both options, a user may terminate a segment named x and later initiate another segment with the same number that also has the same name. In principle, then a user has the power to let a given name take on two (or even more) different meanings, albeit one meeting at a time. He can even cause a reversion to an original meaning (by terminating the second segment named x and reinitiating the original one that was named x). If done carefully, then at no time need there be a risk that a snapped link would be used to address a spurious target.

## 6.5 THE CONFLICT-OF-NAMES PROBLEM

A better understanding of the services and limitations of the SMM can be seen from a study of the double-meaning or conflict-of-names problem which can conceivably plague many a subsystem writer if he is not careful. The problem arises wherever the same reference name is, in different parts of the same process, intended to represent one of two (or more) different files. We illustrate with the following example.

Suppose a subsystem writer, say a civil engineer, wishes to package a set of related segments as a subsystem called "truss" for computing forces in arbitrary structures. For simplicity let us assume this subsystem consists of the principal procedure <truss>, a data segment called <formulas>, and the special routines <cosine> and <arctangent>. Typically, truss users should not be expected to know what procedures <truss> refers to nor where in the hierarchy they are located. Now the difficulty is that some of these procedures may be referenced in <truss> using the same reference names as are other (likely different) procedures of the same names used by the "customer" who has "rented" the use of truss.

We see that the user's process now not only needs to make reference to different segments having the same reference names, e.g., cosine or arctangent, but worse, the user will not be aware of this apparent ambiguity. If, prior to calling <truss>, the user's process has already initiated one version of, say cosine, how can we be sure that, when called, <truss> will link to the cosine it needs? Or, if, after a return from <truss>, the user wants to compute cosine(x), won't the cosine routine that is called necessarily be the one initiated by his process while executing in the <truss> package ? In short, how can the user be sure that each portion of his process will get the cosine of its choice without making an elaborate advanced study of <truss>' "inner works". Clearly, the person who packages and "sells" others on using his truss subsystem must guarantee that all subsidiary procedures, when executing in the truss subsystem, will be properly selected (names paired with the right segment numbers and no interference with name-number pairs needed when executing "outside" the truss package).

This conflict-of-names problem was anticipated in the original Multics design and a general solution for it was not only proposed, but implemented. An SMM was designed which enabled a packager of a subsystem to relate one or more uniquely identified segments to a uniquely identified caller or "parent" segment. Thus if <x> is regarded as the parent, it would be, for example, possible to declare segments <u>, <v>, and <w> as being related to <x> in such a way that when symbolic references, u, v, and/or w are made within <x> (and only within <x>), links are snapped to the intended segments <u>, <v>, and <w>, notwithstanding the possibility that the same reference names u, v, and w had already been used (or would in the future be used) which refer to other segments in the hierarchy. Establishing the required relationship among the segments could be achieved via explicit calls to the SMM. As initially implemented, this SMM proved too costly to use. The current SMM does not now provide the relate facility. Conceivably, however, it could be expanded to achieve a somewhat similar objective. One way that this relating capability could be added would be to expand the KST's (intended) representation of a reference name from what is now basically a 2-tuple, consisting of a context ring number and a name, to a 3-tuple. If the reference name is to be thought of as related to a parent, then the third element of the 3-tuple would identify the parent segment, (e.g., by its segment number); otherwise the third component would be marked null to indicate no parent implied. For example, suppose <x> is unrelated to any other segment, and suppose a first reference was made to it from ring 33. Then its KST representation might be the character string

"***_33_x",

where *** is some fixed length string which by convention means null. Likewise, if <x> is initiated as segment #201, executes in ring 32, and is the parent of certain segments <u>, <v>, and <w>, (whose unique path names are somehow given to the SMM), then their KST names might, when initiated, appear as

"201_32_u", "201_32_v", and "201_32_w".

We have taken a digression to discuss this type of solution to the conflict-of-names problem to plant hope in the breasts of the faithful. Whether such an extension of the present SMM mechanism may eventually prove feasible is not known now. For this reason the remainder of this section is devoted to a discussion of techniques which the subsystem designer can use now, as alternative ways to deal with potential name conflicts.

1. Changing the Reference Names Used within the Package to Increase the Likelihood of their Uniqueness (or changing the ring of the procedure that executes these symbolic references, or both). Clearly, by qualifying the names used (or changing the context rings), e.g., truss_cosine in place of cosine, there will be a drastic reduction in the possibility for name conflict.* Of course, such an expedient, while it will appear attractive to some, is not a general solution. Packagers who elect this approach would do well to publish the reference names that they have used in their packages so that their subscribers may be forewarned of the consequences when using similar reference names in their own programs.

2. Binding the Subsystem Package into One Segment. This approach probably offers the best and simplest solution now available to a subsystem designer. A subsystem designer, after debugging his package, can execute a system command† to bind the segments of his package into a limited number of segments (normally, one for the procedures and one for the data segments, if any). When procedure segments <a>, <b>, and <c> are bound into one segment, their corresponding linkage sections are also bound into one linkage segment. We use the term bind rather than combine to emphasize that in binding the links for all the former intersegment references among the segments of the set being bound, e.g., a call from <a> to <b>, are eliminated in the binding process. The bound package can be executed without further fear of conflict between reference names used within the package and those employed by the user of the package. There is some

*Up to 32 characters will be permitted in a reference name in the file system being reimplemented in 1969, which leaves a good deal of room for qualifiers.

†The binder command is described in BX.14.

disadvantage of binding for the subsystem designer who may wish to maintain his package by periodic updates. The cost of updating may be higher since even minor changes will necessitate rebinding the entire package. But let not the disadvantage just mentioned be construed as a weakening of the argument for binding. Rather, a wise subsystem designer will see added reason for carefully debugging (and then binding) his package before renting it out.

3. Terminating Names (segments) which may have been previously initiated before making new symbolic references with the same names. This approach can hardly be regarded as offering a general solution, but may be useful in special situations. It illustrates one way that the subsystem designer may find himself interacting directly with the SMM.

   To outline this approach we make our example of the truss package more concrete using Figure 6-7 as an aid. Here we assume that the truss system designer is usera3. We also assume that he creates a new director for the truss package, letting a branch directory be called truss_system, and gives everyone E (execute) access to this directory by an appropriate ACL entry. Only the designer has W (write) and A (append) access to truss_system. The designer can later add authorized users to the permission lists of the branches in truss_system as his customers "sign up" for the service. An authorized customer gains access to the truss subsystem by establishing a directory link to <truss> as shown in Figure 6-7. The <truss> procedure would be programmed so that upon entry the working directory is set to truss_system and is reset to the earlier working directory immediately prior to the normal return. The following steps, for example, would accomplish this opjective.

After entry into <truss>

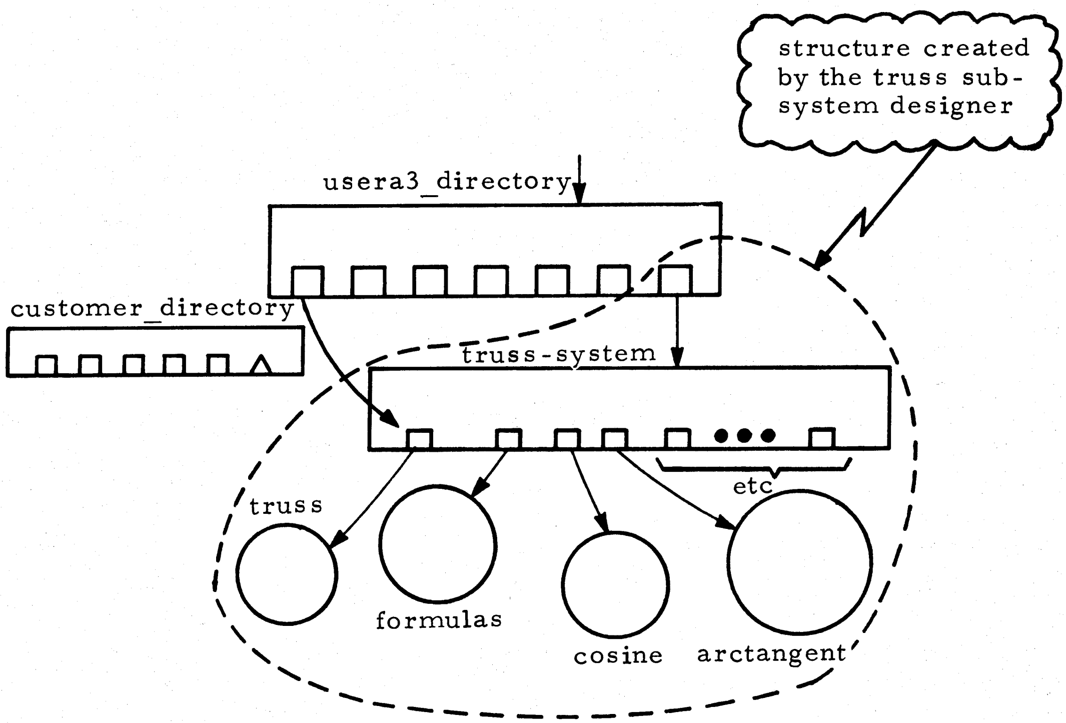
1. Call get_wdir which returns the string of characters in the name for the current working directory (BX.8.12).
2. Save this value in saved_name for use in step 1A.
3. Execute: call change_wdir ("truss_system);

When ready to return to the caller of <truss>

1A. Call change_wdir (saved_name).

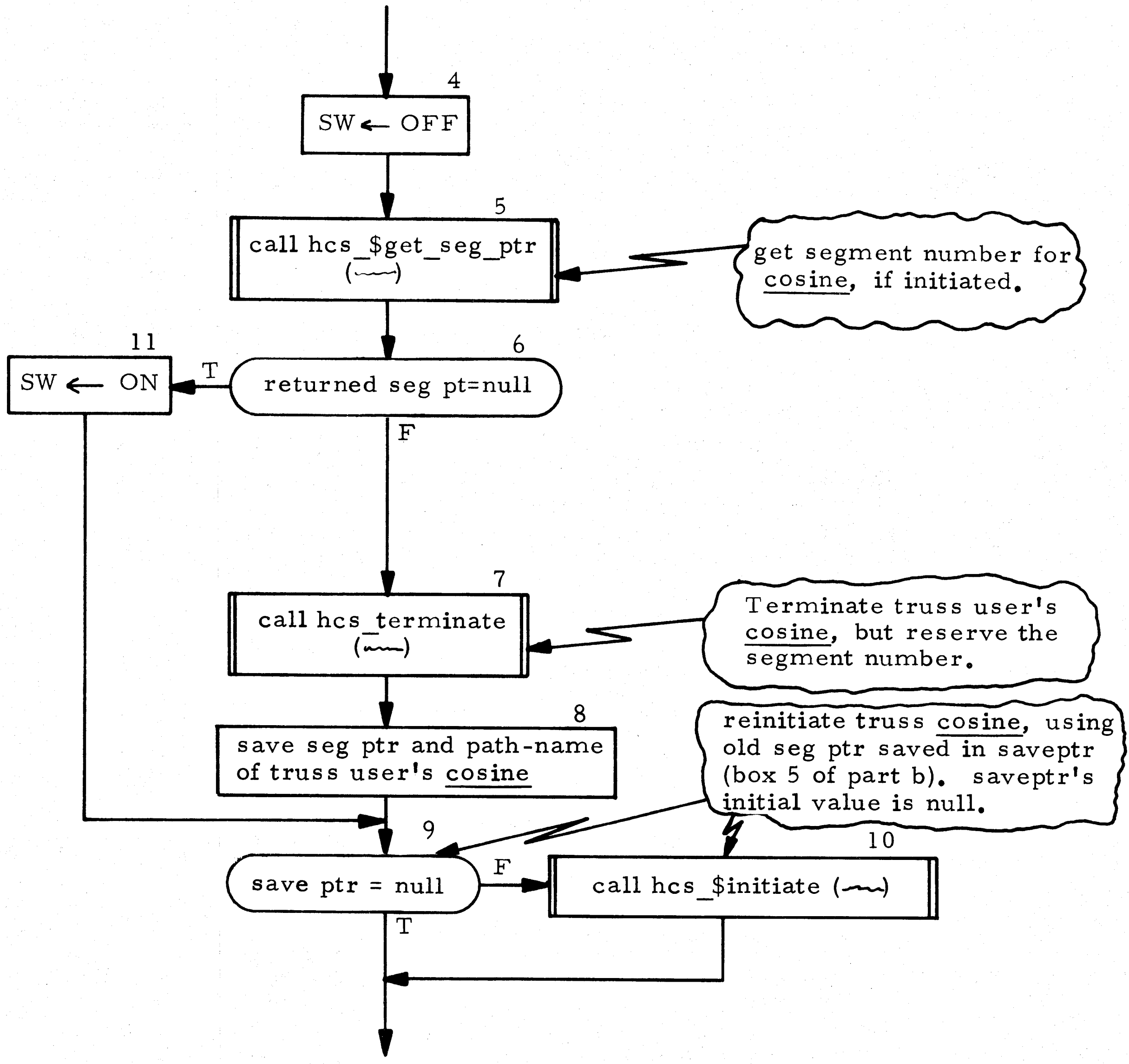
Next, <truss> could attempt to terminate any segments named cosine (likewise, formulas and arctangent) which might already be initiated at the time the package is entered, as illustrated in boxed 5, 6, 7, and 11 of Figure 6-8a. Segments which are terminated, if any, would be those the user of <truss> has initiated previously. The segment numbers for such terminated segments, and their path names would be saved (box 8 of Figure 6-8a) so they could be re-initiated later, when preparing for a return from the truss package, as illustrated in boxes 6 and 7 of Figure 6-8b. The cosine of the truss package would be initiated by normal link fault the first time the package is used, but would be terminated upon exit from the package and its segment pointer saved (boxes 2, 3, 4, and 5 of Figure 6-8b). On the second and all repeated entries into the package the truss cosine would be reinitiated explicitly (boxes 9 and 10 of Figure 6-8a). Similar coding would be required for each name in the package (e.g., formulas and arctangent) for which name conflict is to be prevented in this way.

Observation: One wonders if this costly set of repeated calls on the SMM can in practice be justified.



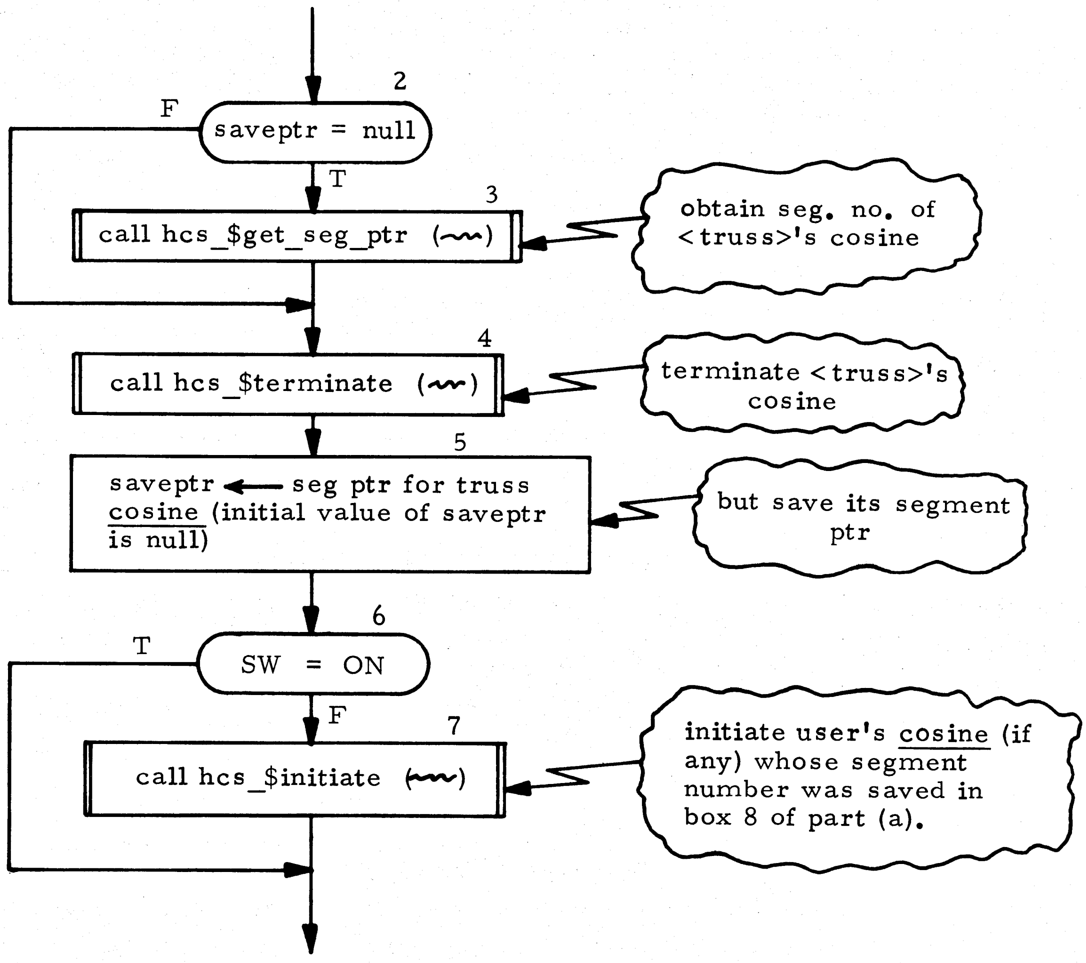
This is for a subsystem package to be used by others.

Figure 6-7. Creating a Separate Directory Structure


4
SW ← OFF
5
call hcs_$get_seg_ptr (~~~)
get segment number for cosine, if initiated.
11
SW ← ON
T
6
returned seg pt=null
F
7
call hcs_terminate (~~~)
Terminate truss user's cosine, but reserve the segment number.
8
save seg ptr and path-name of truss user's cosine
reinitiate truss cosine, using old seg ptr saved in saveptr (box 5 of part b). saveptr's initial value is null.
9
save ptr = null
F
10
call hcs_$initiate (~~~)
T

(a) These would be needed to possibly terminate an old cosine and initiate (as required) a new one upon entering the truss package.

Figure 6-8. Illustrating Direct Calls to the SMM


2
F
saveptr = null
T
3
call hcs_$get_seg_ptr (~)
obtain seg. no. of <truss>'s cosine
4
call hcs_$terminate (~)
terminate <truss>'s cosine
5
saveptr ← seg ptr for truss cosine (initial value of saveptr is null)
but save its segment ptr
6
T
SW = ON
F
7
call hcs_$initiate (~)
initiate user's cosine (if any) whose segment number was saved in box 8 of part (a).

(b) These would be needed to terminate the <truss>'s cosine and reinitiate the user's cosine, if one had been initiated at the time the truss package was entered.

Figure 6-8. Illustrating Direct Calls to the SMM

## 6.6 SEGMENT DESCRIPTOR MANAGEMENT

Initiating a segment, i.e., giving it a KST entry, is only one of the steps necessary for a user's process to gain access to the segment. No segment descriptor word will yet have been constructed for the segment in this process.* Moreover, the segment itself, or more precisely, the referenced page of the segment is very likely not in core at the time the segment is initiated.† Building the SDW and loading the segment (i.e., placing in core the page table for the segment and the required page that is implicit in the user's symbolic reference) are mechanisms relegated to Segment Control. These are tasks to be completed, when necessary, only after the Linker has snapped the link and control has been returned via the Fault Interceptor to the faulting procedure. At this time, the faulting procedure is allowed to resume its attempt to form the target address that is now held in the snapped link. This target will point to a (preset) zero-valued SDW. The hardware then interprets this zero as a missing segment fault. It is this induced fault which will invoke the service of Segment Control to complete the accessing path to the target.

In summary, numerous symbolic references to the same segment may result in an equal number of link faults, each leading to a call on the SMM for a segment pointer. But, only the first of the link faults to the same segment will require that the SMM initiate the segment. Likewise, it is approximately correct to say that use of only the first of the snapped links to a segment will result in a segment fault that triggers construction of an SDW and the loading of the required page table and the desired page.

We would further remark that so long as the segment remains "active," i.e. so long as its page table remains in core, additional references to the same segment can pass successfully through the SDW. These references may, however, incur page faults in the event the desired page is not in core. A page fault induces the loading into core of the referenced page.

The remainder of this section attempts to show how Segment Control constructs the SDW's, as needed, from the information stored in the KST entries. This discussion is provided not so much because of its immediate utility in the design of subsystems, but because it may help some avid reader tie together a number of "loose ends" regarding segment addressing, access control and protection in the Multics

*Of course, if this segment was once initiated but later terminated, the SDW is actually being constructed for a second time.

†The segment would be loaded in the event that some other active process had already initiated and loaded a segment having the same unique identifier.

operating environment. It would be perfectly reasonable to skip this material especially during a first reading.

### 6.6.1 Constructing Segment Descriptor Words after Snapping a Link

We start by picturing the various descriptor segments of a process, i.e., one for each ring in which the process has thus far executed code. At the time segment number i is initiated, the contents of each descriptor segment at the offset = i are (preset) to zero.

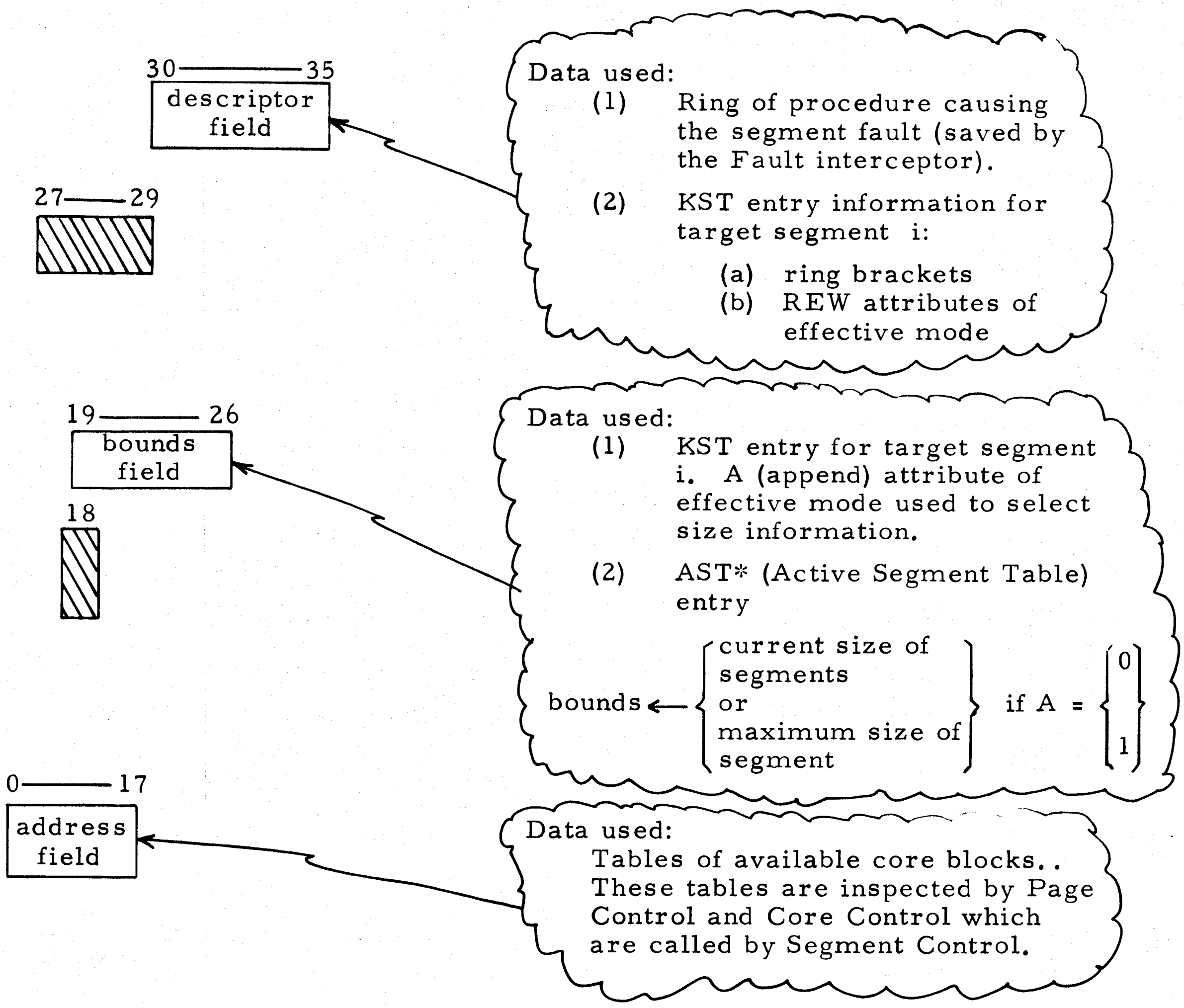
We now consider what happens after the Linker has converted the link pointer to the appropriate its pair and control has returned to the faulting procedure. As mentioned in an earlier paragraph, a segment fault will occur immediately, as a consequence of attempting to form the indirect address implied by the its pair. The segment fault, of course, occurs because the address formation mechanism of the GE 645 will attempt to employ the word at offset = i in the currently-employed descriptor segment. A zero word is interpreted by the hardware as a missing segment fault.

When the Fault Interceptor is again involved, i.e., via the segment fault, it calls Segment Control to construct the proper SDW (Segment Descriptor Word). Figure 6-9 summarizes the sources of information used by Segment Control in constructing as SDW.

Figure 6-10 shows logic exercised by Segment Control in determining the six-bit descriptor field. The primary question that is resolved is: "Was the segment fault due to an inter-ring reference"? If so, the protection rules described in Chapter 4 govern the determination of the descriptor bits (boxes 2, 3, 4, 5, and 6 of the logic). If not, the R, E, W mode attributes of the target (found in its KST entry) govern the determination (boxes 7, 8, 9, 10). Also, as indicated in box 11, a page table is created for this segment, a pointer to it is placed in the SDW, and the reading in of the wanted page is initiated.

Upon completing the SDW, Segment Control returns via the Fault Interceptor to the faulting procedure which can now again attempt to complete execution of the instruction that faulted. Although the SDW is not now zero, it may nevertheless have been coded as one of the faults or potential faults indicated in Figure 6-10.

If the coding corresponds to an inward or outward procedure reference (flow chart boxes 5 or 6), then the executing procedure will fault again. This time the fault handler will be the Gatekeeper. During the course of its work, the Gatekeeper effects a ring change by causing the descriptor base register (DBR) to be reset so



Data used:
(1) Ring of procedure causing the segment fault (saved by the Fault interceptor).
(2) KST entry information for target segment i:
(a) ring brackets
(b) REW attributes of effective mode

Data used:
(1) KST entry for target segment i. A (append) attribute of effective mode used to select size information.
(2) AST* (Active Segment Table) entry

$$\text{bounds} \leftarrow \left\{ \begin{array}{l} \text{current size of segments} \\ \text{or} \\ \text{maximum size of segment} \end{array} \right\} \text{ if } A = \left\{ \begin{array}{c} 0 \\ \\ 1 \end{array} \right\}$$

Data used:
Tables of available core blocks.. These tables are inspected by Page Control and Core Control which are called by Segment Control.

Sources of data used by Segment Control in constructing the descriptor, bounds, and address fields of an SDW for a target segment i.

*The AST is another ring-0 data base. This per-system table is discussed in Chapter 7.

Figure 6-9. Sources of Data used by Segment Control

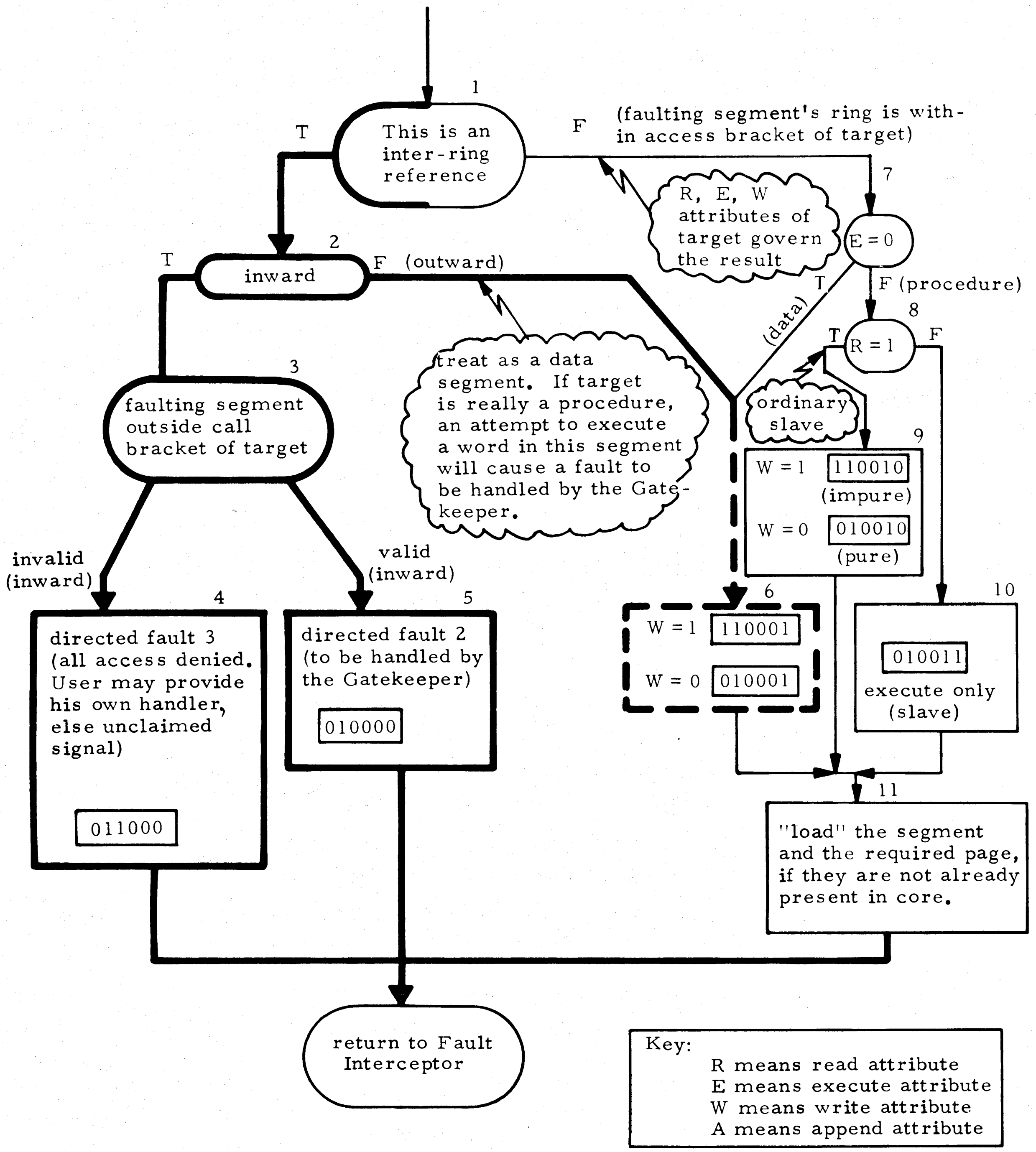

1
This is an inter-ring reference
T
F
(faulting segment's ring is within access bracket of target)
R, E, W attributes of target govern the result
7
E = 0
2
inward
T
F (outward)
treat as a data segment. If target is really a procedure, an attempt to execute a word in this segment will cause a fault to be handled by the Gatekeeper.
T
(data)
F (procedure)
8
R = 1
T
F
ordinary slave
3
faulting segment outside call bracket of target
invalid (inward)
valid (inward)
9
W = 1 110010 (impure)
W = 0 010010 (pure)
4
directed fault 3 (all access denied. User may provide his own handler, else unclaimed signal)
011000
5
directed fault 2 (to be handled by the Gatekeeper)
010000
6
W = 1 110001
W = 0 010001
10
010011 execute only (slave)
11
"load" the segment and the required page, if they are not already present in core.
return to Fault Interceptor
Key:
R means read attribute
E means execute attribute
W means write attribute
A means append attribute

Figure 6-10. Logic used by Segment Control

it now points to the target ring's descriptor segment. Eventually, the Gatekeeper will return control via the Fault Interceptor to the faulting procedure. If the target has never before been referenced in the target ring, then the SDW found at the same offset (=. i) in the target ring's descriptor segment will again be zero and cause another segment fault.

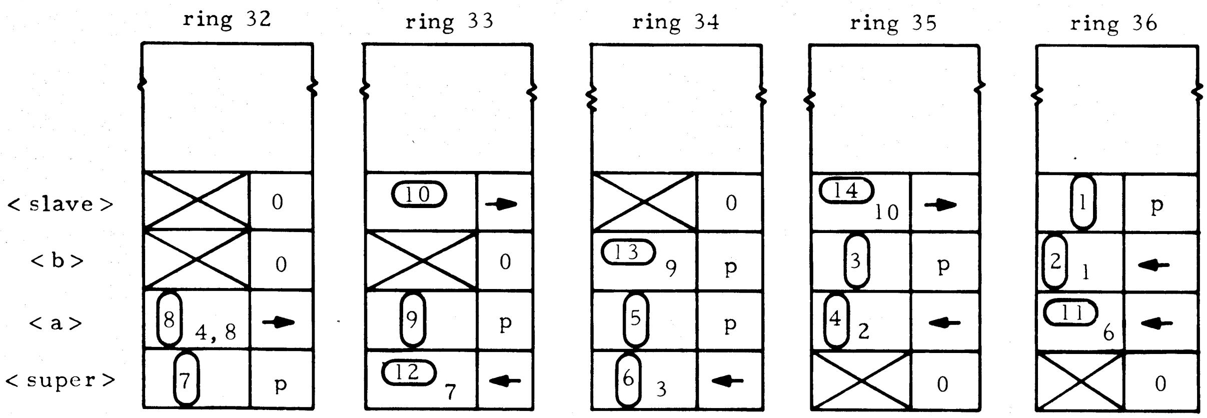
This fault is again handled by Segment Control. The Fault Interceptor will have recorded, as part of the saved machine conditions, the segment number and core location of the SDW causing the fault, and the (new) ring number of this target segment. Consequently, Segment Control is fully able to proceed with the task of forming the new SDW using, of course, the same set of rules (Figure 6-10) as before. (This time, however, the logic in boxes 7, 8, 9, 10, and 11 will be followed.) When control is returned to the faulting procedure, address formation will be allowed to proceed through the newly formed SDW to the now-loaded target.

We are now in a position to picture the evolution of multiple descriptor segments in the multi-ring environment. The SDW's are set dynamically, i.e., one at a time in the rings where they are used, on an as-needed basis.

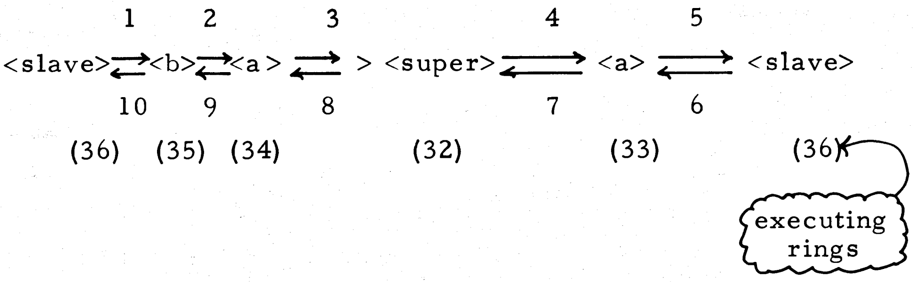
To illustrate how this activity would proceed, we deliberately choose the rather exotic (in fact, unlikely) case first displayed in Figure 4-9. This case involved four procedures, < slave>, < a>, < b>, and < super>, each with different ring brackets.* If < slave> has just been initiated in the process, and if a chain of calls emanates from < slave> to < b> to < a> to < super>, these additional procedures will become initiated in the course of executing this chain.

Figure 6-11 gives a particular hypothetical chain of calls and shows the sequence in which the SDW's would be created via segment faults. Note that the positions of the SDW's in their respective descriptor segments reflect the order in which the corresponding segments are initiated in the process. Each descriptor segment will typically have some SDW's that have not been set. The setting of the SDW's is strictly a dynamic affair, depending upon the particular thread of control that has been followed in the process.

*The straddling access brackets of < a> and < b> cannot be justified in a typical application. They are used here to illustrate the coordination between Segment Control and the Multics protection rules and mechanism which were described in Chapter 4.



Illustrating the sequence in which segment descriptor words are created (circled numbers) via segment faults. Non-circled numbers refer to the sequence of calls and returns as follows:



Assumptions: No prior reference have been made to <a>, <b> or <super>. Moreover, we assume that <slave> has been referenced for the first time by a call from within ring 36. Ring brackets for the segments in this figure are:

(36, 36, 36) for <slave>
(34, 35, 35) for <b>
(33, 34, 34) for <a>
(32, 32, 32) for <super>.

Figure 6-11. The Sequence in which Segment Descriptor Words are Created

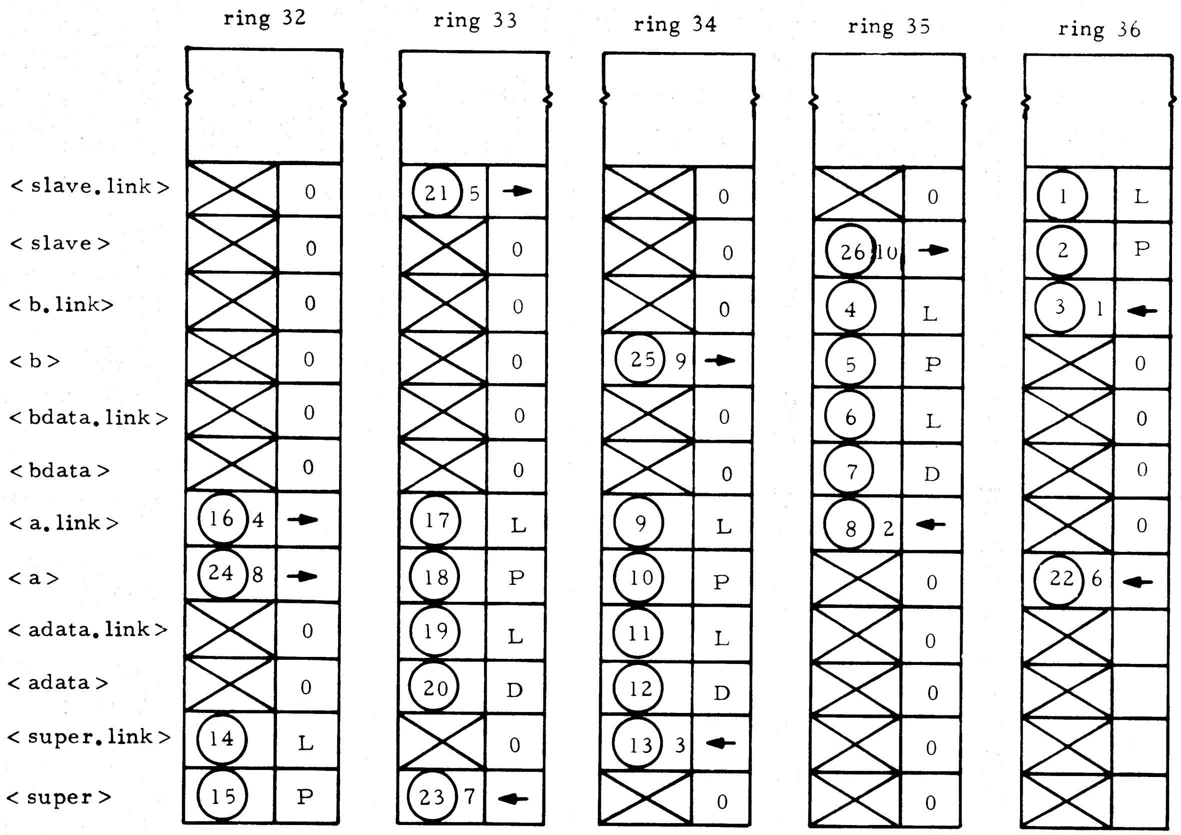
Figure 6-11 is still oversimplified in the following respects:

(1) Normally, when the Linker is invoked to snap a link, it needs to "get its hands on" both the text segment and its linkage segment. For example, on a first reference to a procedure segment < t>, the Linker will ask the SMM for pointers to both < t> and < t. link>. You will recall that to complete a transfer from procedure < a> to procedure < t>, control passes via < a. link> through < t. link> before reaching < t>. This idea was illustrated in Figure 2-12. A quick review of this figure is suggested.

(2) We have disregarded the possibility that the various procedures shown in Figure 6-11 may also make intersegment data references. Linkage faults resulting therefrom would eventually cause segment faults to the referenced data segments and, quite possibly, to their respective linkage segments as well.*

We give in Figure 6-12 a more realistic version of Figure 6-11, taking these omissions into account. In this figure we suggest that < a> and < b> reference data segments< adata> and < bdata>. We also assume for this example that the access brackets for these data segments are, respectively, identical to those for < a> and < b>.

*In general when snapping any links for intersegment references of types 2 or 4 (see Figure 2-10), which are not self references, the Linker, to do its job properly, must get its hands on both the referenced segment and its corresponding linkage segment. In such cases the Linker must use and possibly alter the Linkage section of the target segment before it can complete the link for the faulting segment.



| | ring 32 | | ring 33 | | ring 34 | | ring 35 | | ring 36 | |
|---|---|---|---|---|---|---|---|---|---|---|
| < slave. link > | ☒ | 0 | 21 5 | → | ☒ | 0 | ☒ | 0 | 1 | L |
| < slave > | ☒ | 0 | ☒ | 0 | ☒ | 0 | 26 10 | → | 2 | P |
| < b. link> | ☒ | 0 | ☒ | 0 | ☒ | 0 | 4 | L | 3 1 | ← |
| < b > | ☒ | 0 | ☒ | 0 | 25 9 | → | 5 | P | ☒ | 0 |
| < bdata. link > | ☒ | 0 | ☒ | 0 | ☒ | 0 | 6 | L | ☒ | 0 |
| < bdata > | ☒ | 0 | ☒ | 0 | ☒ | 0 | 7 | D | ☒ | 0 |
| < a. link > | 16 4 | → | 17 | L | 9 | L | 8 2 | ← | ☒ | 0 |
| < a > | 24 8 | → | 18 | P | 10 | P | ☒ | 0 | 22 6 | ← |
| < adata. link > | ☒ | 0 | 19 | L | 11 | L | ☒ | 0 | ☒ | |
| < adata > | ☒ | 0 | 20 | D | 12 | D | ☒ | 0 | ☒ | |
| < super. link > | 14 | L | ☒ | 0 | 13 3 | ← | ☒ | 0 | ☒ | |
| < super > | 15 | P | 23 7 | ← | ☒ | 0 | ☒ | 0 | ☒ | |

Key to descriptor symbols: L - means a segment whose effective mode is REWA (used for linkage segments)

D - data

P - procedure

← - inward call

→ - outward call

Further details in the steps followed for developing segment descriptor words in the sequence of calls and returns:

$$\langle\text{slave}\rangle \underset{10}{\overset{1}{\rightleftarrows}} \langle\text{b}\rangle \underset{9}{\overset{2}{\rightleftarrows}} \langle\text{a}\rangle \underset{8}{\overset{3}{\rightleftarrows}} \langle\text{super}\rangle \underset{7}{\overset{4}{\rightleftarrows}} \langle\text{a}\rangle \underset{6}{\overset{5}{\rightleftarrows}} \langle\text{slave}\rangle$$

(36) (35) (34) (32) (33) (36)

executing rings

Figure 6-12. Further Details for Developing Segment Descriptor Words