Honeywell

SYSTEM INITIALIZATION PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68) **MULTICS**

RESTRICTED DISTRIBUTION

SUBJECT:

Internal Organization of Multics System Initialization.

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set which, when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).

DATE:

February 1975

ORDER NUMBER:

AN70, Rev. 0

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

File No.: 2L13

AN70

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

ILLUSTRATIONS

SECTION I

OVERVIEW

This Program Logic Manual (PLM) describes the initialization of the Multics system. The term initialization, with respect to Multics, is used to describe the bringing up of the Multics operating system. Multics does not require a "system generation," i.e., a creation of a version of Multics tailored to the requirements of a particular installation. Multics tailors itself to installation requirements, as specified on a deck of cards (the CONFIG deck) provided at initialization time (see the Multics Operators' Handbook, Order No. AM81 for details). Thus, the Multics System Tape (MST), which contains the Multics system, can be used at any site. The operational procedure of taking an MST and loading it (via BOS, the Bootload Operating System; see the Bootload Operating System (BOS) PLM, Order No. AN74, for details) is known as a bootload or boot. At the time of a bootload, programs on the Multics System Tape create the Multics environment and read the programs and data on the tape into main memory and virtual memory. Configuration-dependent data is processed and system data bases are initialized. The process of creating the Multics environment is known as initialization.

This PLM describes the procedures, data bases, strategies, and policies used during Multics system initialization. This is to be distinguished from process initialization, which is the initialization of each Multics process shortly after it is created. Process initialization is covered in Process and Processor Control PLM, Order No. AN60. This manual does not cover the initialization of the system control, answering service, and accounting facilities. These facilities are described in System and User Control, Order No. AN66 and System Administration, Order No. AN72. Thus, this manual describes only the initialization of the hardcore supervisor at system bootload time.

A large part of initialization consists of the initialization of various subsystems within the supervisor. In many cases, this initialization is a crucial part of the operation of an individual subsystem and thus is also covered in some detail in the PLM describing that subsystem. This is particularly true of Storage System, Order No. AN61, Process and Processor Control, Order No. AN60, and Reconfiguration, Order No. AN71.

This PLM also covers system shutdown, whose organization and implementation are related to initialization. Shutdown consists of the orderly stopping of a Multics system, either by operator command, or by BOS command following a system crash. The latter is known as an emergency shutdown.

This PLM is organized as follows. Section I gives overviews, and discusses policies and environments not specific to any part of initialization or shutdown, but of interest during all of it. These descriptions are in some sense a collection of information about these policies throughout initialization. Because of its supreme importance throughout initialization, the Segment Loading Table (SLT) is described in Section I. Many of the details in the SLT description are not apparent until later sections. Sections II and III describe the two major phases of initialization. Descriptions of the programs, the building of the environment and overviews are provided. Section IV describes shutdown. Section V is a module-by-module listing of initialization and shutdown modules, providing capsuled descriptions of their function. Some miscellaneous modules are also described.

STRATEGY OF INITIALIZATION

The overall strategy of initialization is that of a "bootstrap" process. That is, the first procedure of initialization runs in an environment devoid of all software assistance. Each new mechanism (segmentation, stacks, symbolic linkage, I/O, interrupts, paging, etc.,) is made operative as soon as possible and then used to enrich the environment in which further mechanisms are made operative. Many mechanisms have subsystems of the supervisor that control them and these subsystems must be initialized before the associated mechanism can be used. The initialization of most subsystems is

accomplished by a call from the initialization driver programs to an entry point in that subsystem devoted to the specific purpose required. Most of these subsystem initializations consist mainly of the setting up of data bases (threading of lists, initialization of arrays, etc.), frequently based upon configuration dependent data specified in the CONFIG deck.

Initialization can be viewed as the loading of the procedures and data bases (in part) of the hardcore supervisor from the Multics System Tape. The programs on the tape constitute precisely enough information to bring a bare hardware system (containing no data other than firmware) to Multics command level and allow a reloading of files to take place. (The previous existence of BOS and the CONFIG deck is assumed.) Some of the segments (procedures and data bases) on the MST ultimately wind up in the Multics Storage System; most do not. All of the hardcore supervisor is on the MST. None of it is retained from a previous bootload. Segments are arranged on the MST in such an order that the earlier segments allow as many mechanisms as possible to be used in loading the later segments. For this purpose, the MST is divided into three collections, to be described later.

Initialization can be viewed as the loading of collection 1, the initialization of collection 1, the loading of collection 2, the initialization of collection 2, and the loading of collection 3. This is a very rough description, meant only to illuminate the use of collections.

Among the last mechanisms to become operative are the traffic control and ring mechanisms. One processor (the bootload processor or bootload CPU) performs all of initialization. The bootload processor, while performing initialization, runs exclusively in ring 0. As traffic control is not operative, it is not meaningful to ask in which process initialization is performed. However, the per-process data bases used by initialization ultimately become the corresponding data bases of the initializer process, Initializer.SysDaemon.z. Thus, it can be said in general terms that the initializer process performs system initialization. The last step in system initialization is the calling of the first user-ring procedure of the initializer (system_startup_), or system control process.

THE SEGMENT LOADING TABLE (SLT)

There are two data bases of paramount importance to initialization. The first of these, the CONFIG deck, describes all configuration-dependent data, including table sizes, various software allocations, interrupt cell, port and channel assignments, and available hardware. The CONFIG deck is constructed by BOS, and is described in detail in the Multics Operators' Handbook.

The other data base is the Segment Loading Table. The SLT consists of two logical parts, implemented as the two segments known as slt and name_table, respectively. The first part, or the SLT proper, consists of some fixed information and an array, indexed by segment number, describing all that is known about each segment loaded from the MST. This information is supplied by the MST segment header, a four-word block supplied by the MST generator (see generate_mst in the System Tools PLM, Order No. AN51), which precedes each segment on the tape. The MST generator derives the information from both the system header file and the segment itself. The information can be modified by initialization procedure in some cases.

As procedures and data bases are loaded from the MST by initialization, they are accessed via the segmentation mechanism. This allows the SLT to be accessed as an array indexed by the segment number assigned at that time. Thus, the SLT is essentially a map of the descriptor segment used by initialization. The segment number assigned is used thereafter by every process to access that segment when in ring 0. This is explained more fully in Section III under "Hardcore and Outer Ring Segment Numbers".

Associated with each SLT entry, or SLTE, is a variable amount of variable-length ASCII-coded information. The information includes names of the segment and a possible access control list (ACL) to be associated with the segment if it is ultimately to go in the Multics storage system hierarchy (see "Branch Creation and Connection" in Section III). To allow the SLT to be accessed as an array, this information is stored in the SLT Name Table, which is in the segment name_table. Pointers in the SLTE connect the SLTE to the array of names for each segment in the name table. These names are needed to allow the various initialization procedures to reference themselves and their data bases by name. Twice in initialization a special linker known as the prelinker runs, resolving as many outward references of initialization procedures as is possible at that time. When we speak of the name of an initialization procedure or data base, it is the name stored in the SLT Name Table to which we refer.

These names are also used as entrynames for those segments that ultimately go into the storage system and are added via the normal storage system name appending primitives (see hcs_$append_branch and hcs_$append_branchx in the Multics Programmers' Manual Subroutines, Order No. AG93, also the Storage System PLM).

The segments used by initialization may be divided into two broad categories: those used only by initialization and subsequently discarded and those that are part of the normal Multics system. The former are known as initialization segments, the latter as supervisor segments. These terms are used only with respect to initialization. The segment numbers assigned to supervisor segments by initialization start at zero; initialization segment numbers start at 400. (All numbers given here are octal unless otherwise specified. Numbers given in English, e.g., thirty seven, are decimal. All numbers in PL/I declarations are decimal.)

Consider the declaration of the SLT.

```
declare 1 slt based (sltp) aligned,

        2 name_seg_ptr pointer,
        2 entry_length fixed bin(18),
        2 first_sup_seg fixed bin(18),
        2 last_sup_seg fixed bin(18),
        2 first_init_seg fixed bin(18),
        2 last_init_seg fixed bin(18),
        2 free_core_start bit(18) unaligned,
        2 free_core_size bit(18) unaligned,
        2 seg (0:8191) aligned,
          3 slte like slte; /*slte declaration given below*/
```

where:

1. name_seg_ptr — is an ITS pointer to the SLT Name Table, word 0.
2. entry_length — is 4.
3. first_sup_seg — is 0, the segment number or the lowest-numbered supervisor segment.
4. last_sup_seg — is the segment number of the highest-numbered (hence the last) supervisor segment loaded.

5.	first_init_seg	is 400, the number of the lowest-numbered initialization segment.
6.	last_init_seg	is the segment number of the highest-numbered initialization segment loaded.
7.	free_core_start	is the address, rounded up mod 64(10) and divided by 64(10) of the first free block of main memory after the permanent unpaged segments. This is explained under "Memory Management" and "bootstrap1."
8.	free_core_size	is the number of whole 64 (10) word blocks (a block must start on a 64 (10) boundary) available between the end of the permanent unpaged segments and the beginning of the temporarily unpaged segments (also to be explained), after bootstrap1 has run.
9.	seg	is the array of SLT entries.

Now consider the SLT entry declaration. Remember for segments that have the header attribute linkage, the linkage section has been stripped off by the MST generator and made into a separate segment. This segment has the names of the main (text) segment, with the suffix link appended to each and, with its header, follows the text segment immediately on the MST.

```
declare 1 slte based (sltep) aligned,

        ( 2 names_ptr bit  (18),    /* word 0 */
          2 path_ptr bit(18),

          2 access bit(4),  /* word 1 */
          2 cache bit(1),
          2 pad1 bit(1),
          2 pad2 bit(6),
          2 wired bit(1),
          2 paged bit(1),
          2 per_process bit(1),
          2 ds bit(1),
          2 dirsw bit(1),
          2 acl_provided bit(1),
```

```
2 pad3 bit(3),
2 branch_required bit(1),
2 init_seg bit(1),
2 temp_seg bit(1),
2 link_provided bit(1),
2 link_sect bit (1),
2 link_sect_wired bit(1),
2 combine_link bit(1),
2 pre_linked bit(1),
2 pad4 bit(7),

2 cur_length bit(9),  /* word 2 */
2 ringbrack(3) bit(3),
2 segno bit(18),

2 pad5 bit(3), /* word 3 */
2 max_length(9),
2 bit_count bit(24)) unaligned;
```

where:

1. names_ptr — is the offset into the name table segment of the name structure for this segment, which is declared below.

2. path_ptr — is the offset into the name table segment of the directory pathname structure (declared below) of this segment if slte.branch_required is on. This is the name of the directory in the storage system hierarchy in which this segment is to be ultimately placed. Only some segments from the MST go into the storage system heirarchy.

3. access — is the first four bits of the access field of the SDW that are constructed by initialization for this segment. These are the Read, Execute, Write, and Privileged bits. This is not the access that goes into any ACL entry in the branch. This is the same access used in the hardcore descriptor for this segment in every process.

4. cache — is the SDW cache bit for the hardcore and initialization SDWs constructed for this segment. If on, the segment is to be allowed in the cache.

5. wired

if this bit and the paged bit (see below) are on in this SLTE, this segment is to have all its pages wired (made nonreplaceable in main memory to page control). The segments pds and pl1_operators_ are special-cased and partly wired. If the paged bit is off, this bit is not meaningful.

6. paged

specifies that the segment is to be made paged at an appropriate time. A paged segment can also be wired.

7. per_process

specifies, if on, that the SDW for this segment, as created by initialization, should not be used by process creation (see the Process and Processor Control PLM for details of process creation) when descriptor segments for new processes are created. By default, all of the SDWs for supervisor segments are put in a new descriptor segment when a process is created.

8. ds

is on in the SLTE for the descriptor segment itself. This flag is used to prevent threading of the descriptor AST entry of the segment (see "The Making Paged of Segments" in Section II).

9. dirsw

specifies that this segment is a directory. Not currently used.

10. acl_provided

specifies that an ACL (access control list) structure, declared below, was supplied by the MST generator for this segment. It follows the pathname structure in the name table segment. This bit is only meaningful when the branch_required bit is on.

11. branch_required

specifies that this segment is to go in the Multics Storage System hierarchy. A directory pathname, for the directory to contain this segment, is pointed to by the path_ptr field (see above).

12. init_seg

specifies that this segment is an initialization segment, to be discarded at the end of initialization. If this segment is a temp-seg, this bit is not on.

13. temp_seg

specifies that this segment is a temp-seg. This is a type of initialization segment that is to be discarded at the first purging of such segments after it has been loaded. See "Memory Management" in this section.

14. link_provided

specifies that the linkage section of this segment has been stripped off by the MST generator and follows this segment on the tape.

15. link_sect

specifies that this segment is the linkage section of some other segment, stripped off by the MST generator. If on, the segno field specifies the segment number assigned to the corresponding text segment.

16. link_sect_wired

specifies that the linkage for this segment, which must be a text segment with the linkage_provided bit on, must be wired. This information is used by the prelinker to determine whether the linkage for this segment should be combined with wired linkage for wired segments or nonwired linkage for nonwired segments.

17. combine_link

specifies if the linkage for this segment, which again must have the linkage_provided bit on, should be combined at all. Linkage segments, in general, are temp-segs, that are combined and then discarded. Some are simply discarded, while others remain as self-standing supervisor segments. See the paragraph on prelinking in Section II.

18. pre_linked

is a flag created and used by the prelinker. If on, it indicates that the prelinker has already processed the linkage section of this segment. It prevents the prelinker from combining linkage twice on a segment.

19. cur_length

is the cur_length attribute specified for this segment in the header, in pages. For segments in collection 2 and paged segments in collection 1, it is used by make_sdw to determine which size AST entry should be allocated for this segment. cur_length is usually exactly enough pages to include as many words as specified by the slte.bit_count field. (This can only be overridden by supplying both the cur_length and bit_count attributes in the MST header). bootstrap1 redefines the bit_count and cur_length fields for the data segments for disk subsystems not part of the configuration being bootloaded, zeroing them. The cur_length (and possibly max_length) fields for certain tables in collections!1 and 2 can also be dynamically changed at bootload time by the TBLS CONFIG card.

20. ringbrack

is the array of ring brackets to go in the branch for this segment if it goes in the storage system hierarchy. For the two segments return_to_ring_0_ and restart_fault_, however, these are the ring brackets that go in hardcore descriptors for these segments. In no other case do these ring brackets go in the hardcore descriptors in any process. These two special cases are necessary for the user-ring fault signalling mechanism (see "Hardcore and Outer Ring Segment Numbers" in Section III and the Process and Processor Control PLM).

21. segno

is the segment number allocated to the linkage section of this segment if the linkage_provided bit is on, or the text segment corresponding to this segment if the link_sect bit is on. This field is

used by the prelinker to locate definition sections and fill in LOT entries. (See the Binding, Linking, and Makespace Management PLM, Order No. AN81, for details on linking). This field is filled in by bootstrap1 and segment_loader, the two segment loading programs, as the text-linkage pairing is determined.

22. max_length

specifies the max_length attribute given in the MST header. This is the maximum length, in pages, to which this segment is allowed to grow. It is ignored in the loading of segments in collection 1. If given, it is used to assign an AST entry of proper size for paged segments in collections 1 and 2. It overrides the cur_length attribute. It is also used to set the max_length attribute in the branches of segments that go in the storage system hierarchy.

The max_length in the branches of the SLT and name_table themselves are special cases, however (in the procedure init_branches), as their own SLT entries do not reflect their correct length at the time that this branch creation is done.

23. bit_count

is the actual length, in bits of the segment. This number is used to determine how much space should be allocated for this segment in main memory if it is ever copied or moved. Furthermore, this is the number set in the branch of any segment that goes into the file hierarchy as its bit count. See cur_length above.

Name Table

```
declare 1 name_seg based (names_ptr) aligned,
          2 pad bit(18) unaligned,
          2 next_loc bit(18) unaligned;
```

where next_loc is the relative address into the segment of the next available location where data can be stored.

The structure for one name is as follows:

```
declare 1 segnam based (namep) aligned,
          2 count fixed bin,
          2 names (50 refer (segnam.count)),
            3 size fixed bin(17),
            3 name char(32);
```

1. count — is the number of names given in this name structure.

2. names — are the structures giving the individual names.

3. size — is the number of significant characters in the name.

4. name — is the actual name, left-justified.

Pathname Structure

If slte.branch_required is on, slte.path_ptr gives the relative address into the name_table segment of this structure.

```
declare 1 path based (pathp) aligned,
          2 size fixed bin(17),
          2 name char (168 refer (path.size));
```

where:

1. size — is the number of significant characters in the pathname.

2. name — is the pathname of the directory in the storage system hierarchy into which this segment is to be placed. Note that only as many words as are needed to contain the significant characters are required.

If slte.acl_provided is on, the ACL structure immediately follows the pathname structure. A description follows:

```
declare 1 acls based (aclp) aligned,
          2 count fixed bin,
          2 acl (50 refer (acls.count)),
            3 userid char(32),
            3 mode bit(36) aligned,
            3 pad bit(36) aligned,
            3 code fixed bin;
```

where:

1. count — is the number of ACL terms provided in this structure.

2. acl — is the array of ACL terms.

3. userid — is the User_id (e.g., Greenberg.Multics.a) of this ACL term.

4. mode — is the access mode for that userid. Currently, only the first three bits (read, execute, write) are defined.

5. code — is part of the user interface to ACL terms and is not used by initialization.

Creation of the SLT

The SLT is created by bootstrap1, the very first program of initialization. The first few entries in it are prefabricated in bootstrap1 and describe the descriptor segment, mailboxes and the fault vector, the processor utility segment,[1] the configuration deck, and the SLT name table themselves. The SLT entries are described in the include file slt_init.incl.alm, also used by the MST checker (see check_mst in the System Tools PLM). There are also a few entries built in this way for initialization

---

[1]The processor utility segment (processor_utility_segment), contains the floating fault vector (see the Processor Manual, Order No. AL39). Multics does not now use the floating fault feature.

segments, namely bootstrap1 and the physical record buffer, the latter used for reading tape during initialization. Those segments whose SLT entries are created by bootstrap1 from this include file (these segments are never loaded, per se) are sometimes referred to as collection 0.

The SLT and name table are permanent supervisor segments. They become paged at the time that other segments become paged. They are not wired. They are eventually put in the storage system hierarchy in >system_library_1. The segment number of the SLT is known to be seven as BOS uses it to find other segments and must start somewhere.

The SLT is also used by some user-ring debugging tools to obtain names of hardcore segments for error messages (see User Debugging and Tracing Tools, Order No. AN79) and by some tools (see the System Tools PLM, and System Dump Analysis, Order No. AN53).

THE ENVIRONMENT PASSED TO INITIALIZATION

The following discussion describes machine state and data given to bootstrap1, the first program of initialization.

The command BOOT is given to BOS to initiate the bootloading of the Multics system. BOS maintains an "image" of Multics core, consisting of all of the core configured into the system not used by BOS, plus a disk buffer area representing the core that is held by BOS. The management of this buffer is described in the BOS PLM, Order No. AN74. The final operation invoked by the BOOT command is the transfer of this buffer into actual core and the transfer of control to a set location.

Multics expects all of core to contain zeroes, except for four items, described later. Thus, the BOOT command zeroes the entire Multics core image.

Multics expects the data contained in the configuration (CONFIG) deck, in its mostly ASCII format as produced by BOS, to reside in location 6000, occupying one page. Thus, the BOOT command copies this information from its storage area within BOS to this point in the Multics core image.

Throughout all of Multics and BOS operation, the page at location 4000 contains a program known as the BOS toehold. This program is described in detail in the BOS PLM. Among its functions are the reading of the disk portion of the Multics core image into core and transfer of control as the last stage of a

BOOT or GO command, and the return of control to BOS and saving of the Multics core image as a result of an orderly or unexpected return to BOS. Multics expects this program to be in this location.

Location 4004 is known to contain a pair of instructions that, if executed in absolute mode, cause a restartable return to BOS, ("restartable" means that BOS can return to Multics at the instruction following the instruction that caused execution of the pair at location 4004.) The occupancy of this page and the location of that pair constitute all of the knowledge that Multics has of BOS. The area containing this toehold is not considered part of the Multics core image and is neither written out when the latter is saved on disk, nor restored when it is read back.

bootstrap1 also expects that the IOM mailbox contains sufficient information to determine the IOM channel number and device number of the tape drive on which the MST is mounted. BOS copies this information into the place in the Multics core image that corresponds to the IOM mailbox.

The final item expected by Multics is the first tape record (past the label and following end-of-file mark) of the Multics tape (MST) to be loaded into the core image. The first program on the tape is bootstrap1. Its first record is loaded so that location zero of this program is at location 10000, absolute. Preceding bootstrap1 on the MST are its MST control words (see generate_mst in the System Tools PLM) and its SLT entry (from the MST header). This data in turn lies on a Multics standard tape record, which has a header eight words long. Thus, the physical tape record has to be loaded into some location lower than 10000. However, to facilitate the loading of the physical record into location 10000, the MST generator strips the first 40 words off bootstrap1 and pads enough words so that when the record is read into location 10000, location zero of bootstrap1 is indeed at location 10000.

Summarizing, the data items expected to be in core at the time control is transferred to bootstrap1 are the CONFIG deck, the BOS toehold, the first record of bootstrap1 itself, and the tape data in the IOM mailbox.

Multics also expects the DATANET 6600 FNP to have been properly bootloaded by BOS. It also expects the DATANET mailbox, the floating fault vector (currently not used) and the bulk store mailbox to be defined in their standard locations.

At BOOT time, BOS modified the CONFIG deck by the addition of a card known as the INTK (for INTaKt <sic>). Its format is

```
INTK bootsw part
```

where bootsw is nonzero if this is a "warm" boot (a hierarchy is present on disk) and zero if cold (not present). part is the name of the partition on disk (normally MULT, but SALV if the salvager is being booted) that Multics is expected to use. This card is looked at by the procedures that must initialize access to the hierarchy (see "Branch Creation and Connection" in Section II).

bootstrap1 expects one and only one processor to be running. It expects to receive control at location 40 relative to its base in absolute mode. It expects that the system controller containing it has the processor on which it is running as control processor. It expects the cache on this processor to be inhibited. It expects index register 2 to contain the absolute location of word zero of bootstrap1, hence it makes no assumptions about where it is, other than that it is on a page boundary in the low-order memory. It expects that index register 0 contains the absolute address of the IOM mailbox and index 1 contains the base address of the interrupt vector, which, on the Model 6180, is always zero. It expects that the tape drive selected by the PCW in the IOM mailbox is in the right data mode and correctly positioned to read the second physical record of bootstrap1.

To facilitate system debugging and problem analysis, initialization interrogates the processor data switches on the bootload processor (the processor that entered bootstrap1) at several times, looking for specific patterns. If these patterns are found at the times they are sought, control is returned to BOS in an orderly fashion, allowing dumping and patching. A subsequent GO command issued to BOS restores control to the point where control left Multics. These patterns (in octal, where "x" represents "don't care") and the points at which they reenter BOS are:

123 4xx xxx xxx	bootstrap1 has just received control from BOS. Only its first record is in core.
123 2xx xxx xxx	bootstrap1 has read itself into core. Nothing else has been read in, still in absolute mode.
123 1xx xxx xxx	bootstrap1 in appending mode, with most of its data bases initialized. Collection 1 not

	loaded yet.
123 x4x xxx xxx	bootstrap1 ready to transfer control to bootstrap2. Collection 1 loaded.
123 x2x xxx xxx	Collection 1 loaded and initialized, ready to load collection 2.
123 x1x xxx xxx	Collection 2 loaded.
123 xx4 xxx xxx	Collection 2 loaded and initialized, ready to load collection 3.
123 xx2 xxx xxx	Collection 3 loaded. Traffic control and I/O not yet initialized.

In the above patterns, the first nine bits (123) identify the data switch settings as "debugging return to BOS." The remaining bits are not mutually exclusive, i.e., more than one may be set, causing many returns to BOS. (These switch patterns can be remembered by the assumption that there are four pairs of values corresponding to stopping before and after loading collections 0, 1, 2, and 3. The 123 is ASCII for S, as in STOP.)

THE INITIALIZATION ENVIRONMENT

The following discussion describes the growth of the Multics environment during initialization. This growth might be viewed as an extraction of relevant information from descriptions of the various parts of initialization. This discussion attempts to answer questions such as "When does paging first become effective?"

Faults and Interrupts

Control is passed to bootstrap1 in absolute mode. The processor is functioning without any segmentation or paging. For the first few instructions, all instructions have the inhibit bit (bit 28) on. No assumptions are made about the contents of the fault and interrupt vectors in main memory. The timer register of the processor is loaded almost immediately with a very large number to prevent timer runouts.

One of the first tasks of bootstrap1 is to mask out interrupts, so that it can cease its use of the inhibit bit. Taking advantage of the inhibit bit could create a lockup fault,

so its use is to be minimized. bootstrap1 determines to which port on the low-order (bootload) memory the bootload processor is attached by interrogation of that system controller with an RGR command generated by an RSCR instruction. The execute interrupt mask assigned to that port is then masked so that no interrupts are allowed. The channel mask in the controller is set fully open as it has not yet been determined precisely what the configuration is.

Once this is done, use of the inhibit bit is curtailed. bootstrap1 now fills in all of the interrupt vector pairs in main memory to ignore interrupts. The fault vector is set to cause a fatal crash on all faults except timer runout, which is ignored. All interrupts are then enabled. Appending mode is soon entered. bootstrap1 eventually transfers to bootstrap2 after the loading of collection 1, which eventually gets to call (through initializer and init_collections) initialize_faults$fault_init_one.

This is one of the earliest calls in initialization as it must take responsibility for fault handling away from bootstrap1. Here, the fault vector pairs for timer and lockup are set to pairs that ignore these faults, as are all interrupt vector pairs. This fault and interrupt ignoring is handled by wired_fim$ignore, storing SCU data at prds$ignore_data. Page faults (directed fault 1) are directed to the proper entry, page_fault$fault, as are segment faults (directed fault 0, directed to fim$primary_fault_entry), and connect faults (wired fim$connect_handler). Segment fault handling must be initialized at this time because it is used in collection 2 initialization before all of the faults relevant to collection 2 have been initialized properly. All other faults are directed to ii$unexp_fault, storing SCU data at prds$sys_trouble_data. Hence, unexpected faults during most of initialization store their SCU data here and this area is the first place to analyze during system problems relating to initialization. The floating fault vector, assumed at location 1020, is initialized to direct floating faults to a handler in fim, which crashes the system. The floating fault feature, under the control of the processor mode register, is currently not used by Multics.

Later in collection 1 initialization, data extracted from the configuration deck is used to determine the assignment of interrupt cells. This is done after system configuration (system controllers, processors) has been ascertained and verified. At this time, initialize_faults$interrupt_init is called. This entry directs all interrupts to their final handlers, in most cases the interrupt interceptor (ii). During interrupt initialization (initialize_faults$interrupt_init), all interrupts

other than the system trouble interrupt are masked off. When this is complete, all legal system interrupts are then unmasked and the channel mask on the bootload system controller set properly. These masks are determined from the configuration data. All unassigned interrupts are directed to syserr$syserr_int. Collection 2 is now loaded. Immediately before the accessing of the storage hierarchy, when bounds faults on directories are possible, all faults are directed to their proper handlers. This is done by initialize_faults$fault_init_two.

Error Handling

All errors, unexpected faults included, encountered by bootstrap1 or the fault vectors set up by it, result in the processor halting (on a DIS instruction) with an error code stored in the accumulator and registers stored in main memory. The identity of these errors must be ascertained from analysis of these quantities. Once initialize_faults$fault_init_one has run, unexpected faults cause a return to BOS with machine conditions stored at prds$sys_trouble_data. Soon after this entry has been called, the procedures to initialize the operator's console under Multics (its data bases, DCW lists, etc.) are called. Note that until interrupts are fully initialized, the operator's console runs without the use of interrupts. As the time zone in which Multics is running has not been established at this point, messages reported on the operator's console until the time zone has been established are generally incorrect (assumed Greenwich Mean Time) in their time designation. initialize_faults$fault_init_one sets some pointers used for clock reading to temporary values, simply to allow clock reading to function without causing problems. The operator's ccnsole logging mechanism is not initialized until the end of collection 1 initialization as this requires the support of the full paging mechanism. After the operator's console has been initialized successfully, most errors detected by programs (configuration inconsistencies, errors normally detected by Multics, etc.) cause a message to be printed on the operator's console and possibly logged. If the error is fatal, i.e., causes a return to BOS, the problem can be analyzed by tracing the call history of syserr, the operator's console manager.

There are several circumstances in which errors detected by programs cause a return to BOS without a message. These include the taking of certain faults when disallowed. These returns to BOS are accomplished via the system trouble interrupt, a software-defined interrupt that causes all processors to halt, except the bootload processor, which returns to BOS. Again,

except the bootload processor, which returns to BOS. Again, machine conditions are stored at prds$sys_trouble data, allowing identification of the program that sent the system trouble interrupt.

Difficulty in initializing the operator's console also causes a return to BOS without a message.

Segmentation and Paging

bootstrap1 receives control in absolute mode. As soon as it has read itself in (remember that BOS loads only its first record), a descriptor segment is set up. This descriptor segment, as all segments set up by bootstrap1, is a contiguous, unpaged segment. It is set up off of the end of bootstrap1. SDWs are created in this descriptor segment to describe the descriptor segment itself, bootstrap1 itself, the interrupt vector (including the fault vector), the DATANET 6600 FNP mailbox, the bulk store and IOM mailboxes, the configuration deck as passed by BOS and the floating fault vector. The SLT and the SLT name table are then laid out following the descriptor segment. They are unpaged segments and descriptors are made to describe them. The physical record buffer, a segment used as an I/O buffer for initialization's reading of the MST, is laid out following this. The physical record buffer and bootstrap1 are initialization segments--all of the rest are permanent (supervisor) segments. Appending mode is then entered. As segments are loaded by bootstrap1, descriptors are constructed for them. All of these segments are unpaged. The first paged segment to be constructed is the SCAS, or System Controller Addressing Segment. A segment named scas is loaded by bootstrap1 from the MST. It, like all others at this time is unpaged. The procedure scas_init, running at collection 1 initialization, constructs in this segment a page table, eight entries long, for a segment. This segment has the zeroth word of its nth page being the zeroth word of main memory in the system controller on processor port n (starting from zero). The "unpaged" bit of the SDW for this segment is turned off and the bound set to the minimum, 16(10) words. This segment is used for addressing system controllers for functions other than storing and retrieving data, specifically the RSCR and SSCR instructions. The page table for this segment is completely outside of the domain of page control, which is unaware of the existence of this page table. The areas of main memory pointed to by the scas page table are not reserved--they are used as normal pages of main memory, subject to other constraints. The SCAS is discussed more fully in "Configuration Initialization" in Section II.

The procedure initialize_dims creates the first real paged segment. The File System Device Configuration Table (FSDCT), must be accessed if this is a warm bootload or created if a cold bootload. (See the Storage System PLM for details on the use of FSDCT.) An AST entry containing a page table initially filled with null addresses is allocated. A descriptor describing a paged segment, using this page table, replaces the descriptor for the zero-length segment fsdct. At this time, page control and its I/O routines have been sufficiently initialized so that page faults can be taken. Nevertheless, pages cannot yet be withdrawn from or returned to the FSDCT as the latter is not yet wired, and a page fault during a page fault would result if an attempt were made to access it. If this is a warm boot, it is defined that the first page of the FSDCT resides on record 0 of the MULT partition of the master device (the device with the lowest device ID that has a MULT partition.) This device address is inserted into the address field of the zeroth PTW of the FSDCT. It is defined that a file map (array of device addresses) for the FSDCT begins somewhere on its zeroth page. This file map is copied from the FSDCT into its own page table, other than the zeroth PTW. This access to the FSDCT causes a page fault, but there are no other pages in core that could get written out (causing deposits to the FSDCT if zero) and no page faults have been taken on pages with null address, causing no withdrawals from the FSDCT. As soon as the FSDCT is in core, it is wired and deposits and withdrawals can now function. Paging may be said to be operative at this point, although none of it is going on and only one legitimate (non-SCAS) segment is paged. In the case of a cold boot, the FSDCT is created from scratch. Disk space is allocated for all pages of it, withdrawing it from the FSDCT being created to avoid later withdrawals by page control. It is wired in this case too.

The procedure make_segs_paged (formerly update_sst_pl1) is responsible for making the segments that should be paged into paged segments. This is true only for collection 1 segments, as segments in collections 2 and 3 are always paged. This procedure obtains an Active Segment Table Entry (ASTE) and page table for a new descriptor segment, that will be paged. This descriptor segment will be used by initialization and the initializer process from that point on. This procedure obtains ASTE/page tables for all of the segments to be paged. It copies the unpaged segments (through an auxiliary procedure, privileged_mode_ut$swap_sdw_in_use) into their new paged incarnations. The new SDW for the paged segment replaces the SDW for the unpaged segment in both the current (constructed by bootstrap1) descriptor segment and the descriptor segment being built (the paged one). More will be said about this in the "Main Memory Management" in this Section and in "The Making Paged of

Segments" in Section II. Finally, the SDW of the new descriptor segment is loaded into the DBR. Paging and segmentation are fully operative at this point.

At the time collection 2 is being loaded, an ASTE/page table is allocated for each segment before it is loaded. The size of this page table, like those in collection 1, is determined by the procedure make_sdw from the cur_length and max_length attributes in the SLTE of the segment, which precedes the text of the segment on the MST. The tape reading routine (tape_reader) copies the segments being loaded directly into the segment being constructed. SDWs with correct access information are placed in the descriptor segment after the particular segment is loaded.

The initiation of segments and segment faults do not occur until the initialization of collection 2. The procedure init_root_dir makes a call (to initialize_kst) to set up the KST of initialization (to become the KST of Initializer.SysDaemon.z) to be able to initiate segments. The root is initiated and a segment fault occurs on it. More is said about this under "Root Directory Initialization" in Section III, and the Storage System PLM. Segment faults can now be taken on all segments and branches can be appended and initiated via the normal storage system interfaces. Thus, the segments in collection 3 are loaded by simply appending branches to the storage system and copying segments from the physical record buffer, piece by piece, into the newly created segments. Access is set to allow this copying and set appropriately afterwards.

Branches must be created for some segments in collections 1 and 2. However, since segment faults taken by processes look to the branches of segments to find their AST entries, if active, collection 1 and 2 segments that are to have branches (be in the hierarchy), must have ASTEs and thus page tables and therefore must be paged. This requires many of those segments in collection 1 to be paged. Other than the FSDCT, shutdown stack, Paging Device Map Segment, and PRDS, all of the wired paged segments in collection 1 are paged for this reason.

The hardcore supervisor never takes a linkage fault. The initialization of the system search rules (thus initializing the linker mechanism) is one of the very last things done by system initialization.

The PL/I Environment

bootstrap1 runs in privileged mode since issuing I/O instructions is one of its functions. It uses pointer registers to point to the SLT, Name Table, fault vector, physical record buffer, IOM mailbox, configuration deck, the descriptor segment, and each segment being loaded. It uses no stacks and all calls are made via index registers, i.e., TSXn instructions. It is impure; it modifies its own code and data. It makes no external references as there is no program to resolve such links. It does not even have a linkage section. It remembers the segment number of "interesting" segments as it passes them in loading.

bootstrap1 passes control to bootstrap2 once collection 1 is loaded. Among the information passed with control is the segment number of the segment pds, which will be used as a stack, and the segment number of the SLT manager, an initialization program that can resolve a segment name into a segment number from the SLT. As both bootstrap2 and its linkage section are initialization segments, the former precedes the latter immediately on the MST, and the fact that their segment numbers are contiguous is known to bootstrap2. Thus, bootstrap2 loads the linkage pointer register with a pointer to the base of the segment whose segment number is one greater than the segment number of bootstrap2, the latter being determined with an EPAQ instruction. bootstrap2 now establishes a stack frame on the segment pds, (known as "the PDS"). Multics standard calls can now be made, but not through links, as these have not been snapped. Some information in the stack base of the pds is initialized. Its pointer to the signalling procedure is initialized to point to segment -2, word 2 which would cause a fatal process error. The SLT manager is now called to ascertain the segment number of the prelinker. This call is made based upon the segment number passed by bootstrap1, who noticed and remembered it as this segment was being loaded. As no links have been snapped, all of the procedures that run before prelinking (including bootstrap2, but not bootstrap1) are called via transfer vectors at their zeroth through nth words, different offsets corresponding to different functions. Once prelinking is complete, these procedures are called in the normal PL/I fashion. Among the information passed to the prelinker by bootstrap2 is the segment number of the SLT manager. The prelinker (the programs pre_link_1 and pre_link_2) attempts to snap all links in all hardcore linkage sections, which are conditionally (based on the SLT) combined into combined linkage segments. There is one combined linkage segment that always remains wired and one that is paged and unwired. Again, the decision to place a linkage section in one or the other is

based upon SLT bits. A Linkage Offset Table (LOT), is created. (See the Binding, Linking, and Makespace Management PLM, Order No. AN81, for details on the LOT.) Once the prelinker returns to bootstrap2, symbolic calls can be made. bootstrap2 initializes pointers in the base of the PDS to point to various operators within the PL/I operator segment and PL/I programs can now be used (all programs were assembler-coded up to this point). The call-push-return mechanism is fully operative, and the PDS is being used as a stack (it is unpaged now and thus may not grow, but it may grow after being made paged) and all symbolic references capable of being resolved have been resolved. Thus, the program initializer is called (not transferred to). Signalling and the condition mechanism are not operative. All attempts to signal cause the process-terminating pointer to be indirected through, causing the system to terminate operation (attempts to terminate the initializer process are always fatal to the system). After collection 2 is loaded, it is prelinked to and from collection 1 and itself. initialize_fault_$fault_init_two sets this pointer to the normal signalling procedure, signal_. This is the earliest time that signalable faults are allowed.

The PDS is used as a stack as soon as stacks are used at all. It becomes paged when other segments become paged, creating a unique problem involving the return from the segment paging routine which will be discussed under "The Making Paged of Segments" in Section II. The PRDS is used as a stack during interrupts and page faults, as in normal operation. It too becomes paged at the time that segments are made paged. The stack frame laid down by bootstrap2 remains on the PDS until the program initializer finally calls out (via init_proc and gate_init) to ring 1. Shutdown uses a special stack, shutdown stack, early in emergency shutdown or for the wired portion of normal shutdown.

Traffic Control and Rings

Initialization runs in the address space that is to become the address space of the Initializer process, Initializer.SysDaemon.z. Until traffic control is initialized, after the loading of collection 3, control never leaves the initializer process. If any event (specifically, a disk page fault) must be waited on, a special loop in the program wired_fim is entered by the traffic controller. This loop waits for a flag to be set by a routine that is called when an attempt is made to notify an event, invoked by an interrupt. These special handlings are done because the flag tc_data$wait_enable is zero until traffic control is initialized. No directories should ever

be locked. Idle processes do not exist or run, and the body of the traffic controller (scheduler) is never entered until traffic control has been initialized by the procedure tc_init. Initialization runs in hardware ring 0, not leaving this ring until init_proc calls system_startup_ in ring 1, after all of the hardcore has been initialized.

I/O Management

Input/Output during initialization consists of tape reading, operator's console writing, and paging. bootstrap1 has a physical tape reading routine in its first physical tape record. This routine is initialized to read the MST, from data left by BOS in the IOM mailbox. It sets up LPWs, DCWs and PCWs to read the MST, and issues connects to the IOM. Once bootstrap1 has been read in, a more sophisticated tape reading routine within bootstrap1, which is knowledgeable about the format of Multics Standard Tapes, including error retry conventions, is used to interface to the simpler routine. The sum of these two routines reads the segments and SLT headers of collection 1, using the segment physical_record_buffer as a single record tape buffer. The smaller routine that has read bootstrap1 is not knowledgeable about these things, hence, bootstrap1 must be written on the MST without error.

During collection 1 initialization, the IOM manager and its data bases are initialized. This is the first initialization after the first fault initialization (see "Faults and Interrupts" earlier in this section). The operator's console is initialized next, without the logging facility. It uses the IOM manager to perform physical I/O. The clock reading and interrupt mechanisms for this work through interim methods that have already been described.

Collection 2 is loaded by a program called segment_loader, invoked by the program initializer. This program calls a tape reading package called tape_reader to read the MST. This program is again knowledgeable about Multics Standard Tapes. It uses a program called tape_io to perform physical I/O via the IOM manager. A device index is assigned to the bootload tape drive/channel at the time this package is initialized, immediately before the loading of collection 2. Collection 3 is loaded by a program called load_system (see "Memory Management" in this section), which also utilizes this tape reading package (tape_reader and tape_io),. After collection 3 has been loaded, the bootload tape drive is rewound via a call to tape_reader$final.

Paging I/O is initialized by the individual I/O routines for the storage system devices (disks, bulk store, etc.). These routines are called at initialization entry points by initialize_dims, during collection 1 initialization. This happens after both the IOM manager and interrupts are initialized, but before paging is operative. These routines report to the IOM manager for device index assignment (except the bulk store control routine, as the bulk store interfaces directly to the system controllers, and hence, does not use the IOM). This initialization of communication with the IOM manager (see the Supervisor Input/Output PLM, Order No. AN65, for details on the IOM manager) includes communication of the identity of the interrupt handler of the routine, and base address for DCW lists. These routines are fully operative after this reporting has been done.

Other I/O device control routines (teletypes, I/O interfacer) are initialized after initialization of traffic control which follows the loading of collection 3. These routines are fully operative after initialization. The Network software is initialized by a call from an outer ring, via a gate, if a Network attachment is present, and is thus not part of system initialization.

Communication with the DATANET 6600 FNP needs no initialization. This processor is bootloaded by BOS. The initializations performed by Multics at collection 2 initialization time, in the module dn355_init, consist solely of setting up some data for the DATANET communication routines. In pre-24.4 systems, this is done at collection 1 initialization time.

MEMORY MANAGEMENT

The following discussion covers the development and use of different strategies for manipulating and managing main memory and virtual memory during initialization.

Collections

As has been described, the Multics System Tape (MST) is divided into three collections of segments. The segments are separated by special control words known as collection marks, which are recognized by bootstrap1, segment_loader and load_system, the three segment loading programs. The significance of the three collections is as follows. There is much more data on the MST than can fit into main memory at any

one time. Hence, much of it will have to be loaded into virtual memory, i.e., loaded in the presence of an operative paging environment. Thus, a goal of initialization is the establishment of paged segmentation at the earliest possible time. Collection 1 contains precisely those programs that are required to accomplish this. All segments that are unpaged, as opposed to paged and wired, appear in Collection 1. This is because only bootstrap1 can allocate unpaged, contiguous core in this way. Before System 24.4, the ability to wire segments in Collection 2 did not exist, and all wired segments were in Collection 1. Collection 2 contains all of the rest of the hardcore supervisor, i.e., all of the rest of the programs that will run in ring zero, and must be prelinked to programs in ring 0. The segments are loaded by the program segment_loader directly into the virtual memory. Collection 3 consists of programs that are not part of the supervisor. These programs constitute precisely enough of the system control and user environment to allow a reload of the storage system hierarchy to be performed. All of the programs in collection 3 are loaded directly into the hierarchy by the program load_system.

Supervisor, Init and Temp Segs

As the first two collections are loaded, programs that will be used only by initialization and programs that will remain as part of the Multics supervisor are added to the virtual memory. These segments are known as initialization segments and supervisor segments, respectively. Each of the first two collections contains both types of segments. Among the initialization segments, many are used only once (e.g., scs_init, which initializes configuration-dependent data concerning port assignment and interrupt cells), and many are used more than once (e.g., tape_io, which reads the Multics System Tape). Thus, a further subdivision is made within initialization segments: temp segs are segments that are to be discarded at the first opportunity following their use, and init segs proper, that are to be discarded at the end of initialization. Discarding these segments frees the AST entries they utilize, increasing the AST pool, and the disk and bulk store storage that they may occupy. Furthermore, it removes their SDWs from the descriptor segment that will belong to the initializer process. In order to facilitate the use of this mechanism, many of the temp segs that will be discarded after collection 1 are bound into a single bound segment, bound_temp_1. Temp segs to be discarded after collection 2 (those, obviously, are loaded in collection 2) are in bound_temp_2. Init segs loaded in collection 1 are in bound_init_1, those loaded in collection 2 (used for loading collections 2 and 3, and initializations after collection 3 has

been loaded, e.g., the traffic controller) are in bound_init_2. Most linkage sections, stripped off of their text segments by the MST generator, are temp segs. The keywords init_seg and temp_seg in the MST header specify the assignment of these attributes to segments.

There are seven lists of ASTEs in the Active Segment Table (AST). There are four lists used by Multics while running, for the allocation and deallocation of ASTEs for segments on which segment faults have been taken. These correspond to the four sizes of AST entries, 4, 16, 64, and 256 (decimal, page table size in words.) A fifth list is called the hardcore list. AST entries for supervisor segments loaded in collection 2, and supervisor segments from collection 1 that obtain AST entries at the time that segments are made paged, are put on this list. However, those supervisor segments that ultimately go in the storage system hierarchy are not put on this list, but in one of the four normal lists. This allows shutdown to process these segments when other active segments of the storage hierarchy are deactivated and/or have their branches updated. Shutdown also uses the hardcore list to delete (free the disk storage of) the hardcore supervisor segments. Clearly, the code that does this and runs after this must not delete itself. Hence, all wired, paged, supervisor segments appear on no AST list in 24.4 and later systems. Most of shutdown, however, is unpaged code. A sixth AST list is maintained for init_segs. AST entries allocated for init segs made paged in collection 1 initialization or loaded in collection 2 are put on this list. At the end of initialization this list is traversed, all of these entries are freed, and their segments deleted. The seventh list is the list of ASTEs of temp segs. Temp segs made paged in collection 1 or loaded in collection 2 have AST entries on this list. Before collection 2 is loaded, and again before collection 3 is loaded, this list is traversed and AST entries and secondary storage freed. Clearly, segments that are not paged and hence have no AST entries, occupy main storage permanently (if they are still not paged after segments are made paged.) The program delete_segs is responsible for traversing AST lists and deletion of segments (for a description of delete_segs see Section V).

Some special paged segments have their AST entries threaded out of any AST list to prevent both deletion or branch updating, which happens on the other lists. These segments are the PRDSs of processors, the shutdown stack, and the FSDCT, all of which must be used at shutdown time as segments are being deleted, and have no branches. The root is special-cased by shutdown to avoid deletion and branch updating.

Main Memory Management

Figure 1-1 shows the layout of main memory as bootstrap1-1 receives control from BOS. The only information in main memory is the CONFIG deck, the BOS toehold, the information identifying the bootload tape drive and channel in the IOM mailbox and the first record of bootstrap1. The fault vectors, floating fault vectors, DATANET 6600 FNP and bulk store mailboxes are present, but contain no valid information.

bootstrap1 proceeds to read itself in. It then lays out its descriptor segment, the SLT and SLT name table, and the physical record buffer directly after its own text. Descriptors are made to describe the mailboxes, CONFIG deck, and fault vectors. Figure 1-2 describes the layout of main memory at this time.

bootstrap1 now loads collection 1. As you will recall, all segments are unpaged at this time. Supervisor segments that are to remain unpaged are loaded following the end of the physical record buffer. This is where they will remain throughout Multics operation. They are loaded contiguously, one after the other, in ascending address order in main memory. Keep in mind that the mailboxes as well are permanent unpaged supervisor segments, but they were created as segments in preassigned location by bootstrap1 before collection 1 was loaded. All other segments loaded by bootstrap1 (including init and temp segs, and segments in collection 1 that are to be made paged) are loaded starting at the high-addressed end of available memory. They are allocated contiguously, one after the other, in descending address order in main memory. When all of these segments have been loaded, the starting address and length of the unused core remaining are copied in to the SLT (see the SLT discussion earlier in this section) for the later initialization of the pageable memory pool. Figure 1-3 now shows the layout of main memory.

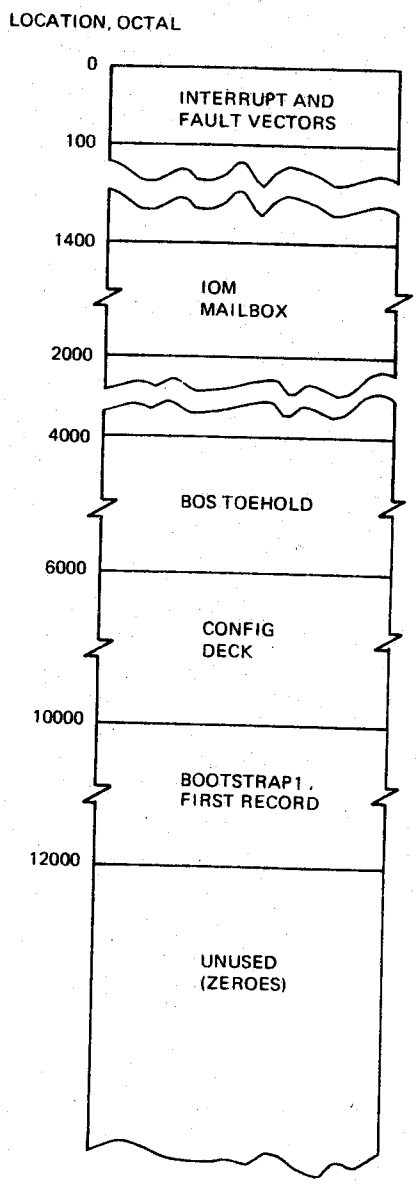

Figure 1-1

Main Memory as bootstrap1 Receives Control

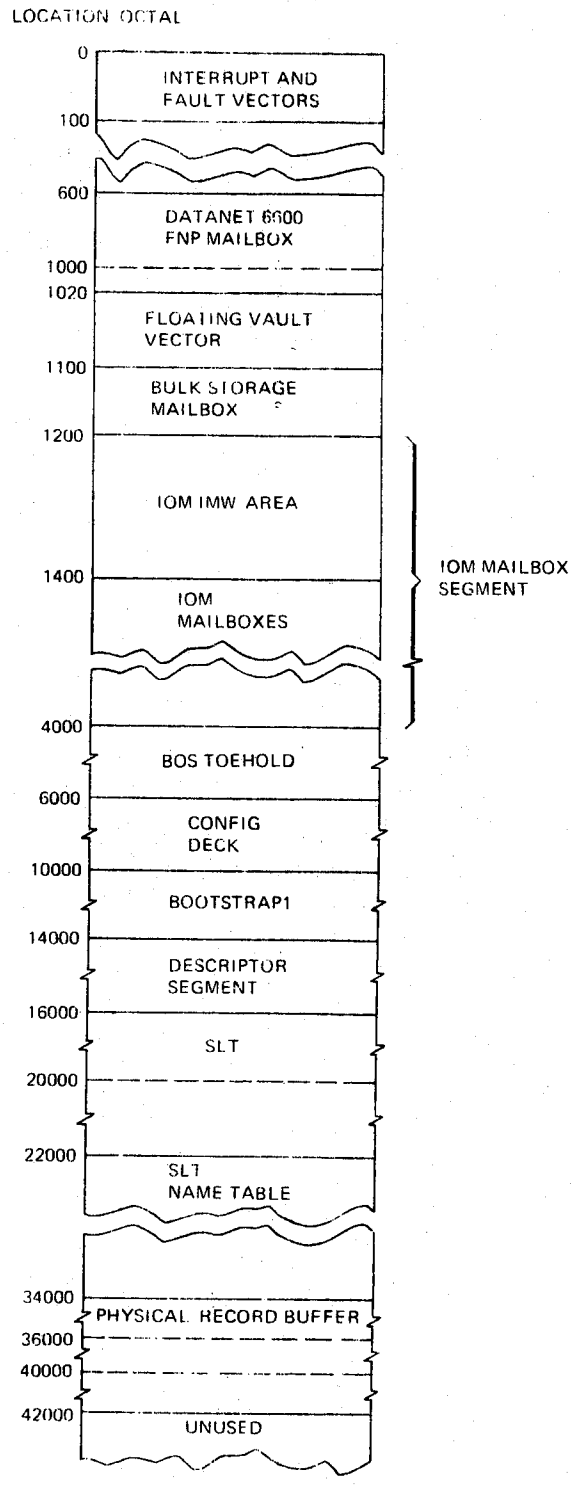

Figure 1-2

Main Memory Before Loading Collection 1

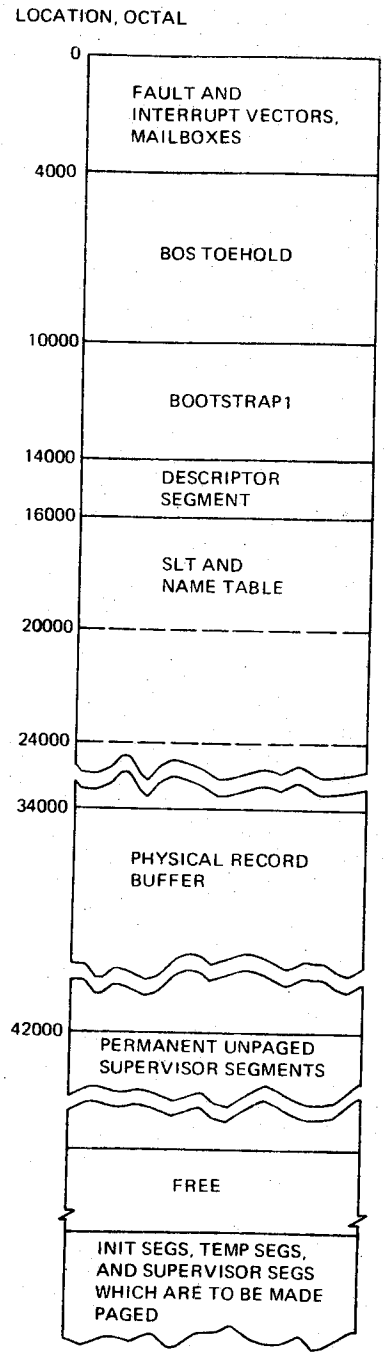

Figure 1-3

Main Memory After Loading Collection 1

Collection 1 is now initialized. All of the segments now existent are unpaged, and hence, no page faults are taken until creation of the FSDCT segment. Before this happens, though, the procedure init_sst runs. This procedure creates the core map (see the Storage System PLM, Order No. AN61) among other data bases, and creates a core used list, consisting of all of the core blocks that lie entirely within the free region left by bootstrap1. Thus, the first few page faults on the FSDCT and the new paged descriptor segment are resolved in this area. If there is not enough room left to resolve these page faults, Multics will crash during initialization.

Once there is a minimal core used list, and paging is operative, the procedure make_segs_paged copies each of the segments at the high end of main memory (the temp, init, and paged supervisor segs) into paged segments, for which it has asked make_sdw to fabricate AST entries based upon the SLTE information available. These segments are copied in descending address order, starting at the high end of memory. As each full page is copied, i.e., one core block worth of information from the high end of memory has been copied into paged segments, the core block is freed, i.e., added to the core used list (the pageable memory pool). During this operation, the amount of pageable memory increases due to this freeing. The occasional wiring of paged segments tends to decrease the amount of available memory. If at any time, the amount of available pageable memory becomes zero, Multics will crash. The time when the minimum amount of pageable memory is available during this operation is known as the "core high water mark" of initialization.

As each of these segments is successfully copied, the descriptor for the new paged segment replaces the descriptor for the unpaged segment in both the old (unpaged) descriptor segment and the new (paged) one. This allows the main memory occupied by the old unpaged copy to be freed immediately. The configuration deck, the SLT, and its Name Table are also copied into paged segments. As they are not contiguous with the segments at the high end of main memory whose memory is freed sequentially, their main memory is not freed at this time. After all of this copying is done, a procedure called collect_free_core runs. This program does a marking-type garbage collection of main memory. All core blocks found to be not in the core used list, but to contain a word of an unpaged segment, are marked. The entire core map is then scanned, and any core block which is marked is unmarked. Any core block that is found unmarked, and is in a currently configured (ON) system controller is added to the core used list. This frees the core formerly occupied by bootstrap1, the SLT and SLT name table and the physical record buffer before they were

made paged, and any other pages never freed (e.g., the page at 2000). The unpaged descriptor segment is then freed, after the new one is in use.

One may note that the physical record buffer becomes paged, and is wired. The initialization tape reader, tape_io, constructs DCW lists that are cognizant of the fact that this segment is not contiguous in memory.

From this point on, main memory control is completely under control of page control. Figure 1-4 shows the current layout of main memory. The unpaged supervisor segments remain in place throughout Multics. Wired as well as unwired segments share the pageable core pool. The deletion of init_segs removes the physical record buffer from memory (unwiring and deleting it).

Shutdown deletes segments, but has no particular effect on main memory allocation.

SUMMARY OF INITIALIZATIONS CALLS

This subsection is a summary of all of the calls made by bootstrap2, initializer, and init_collections, the latter two being simple call dispatchers. This is intended to give an overview of the sequence of initialization before we describe these procedures in detail in the next two sections.

bootstrap1 transfers to bootstrap2, collection 1 having been loaded.

call slt_manager	ascertain segment number of prelinker.
call pre_link_1	prelink collection 1.
call initialize_faults$fault_init_one	sets up interim fault handling, most faults set to be fatal. Page and segment faults legal. Interim clock reading set up.
call iom_data_init	IOM manager initialization. Channel tables and overhead channels set up. Mailboxes initialized.

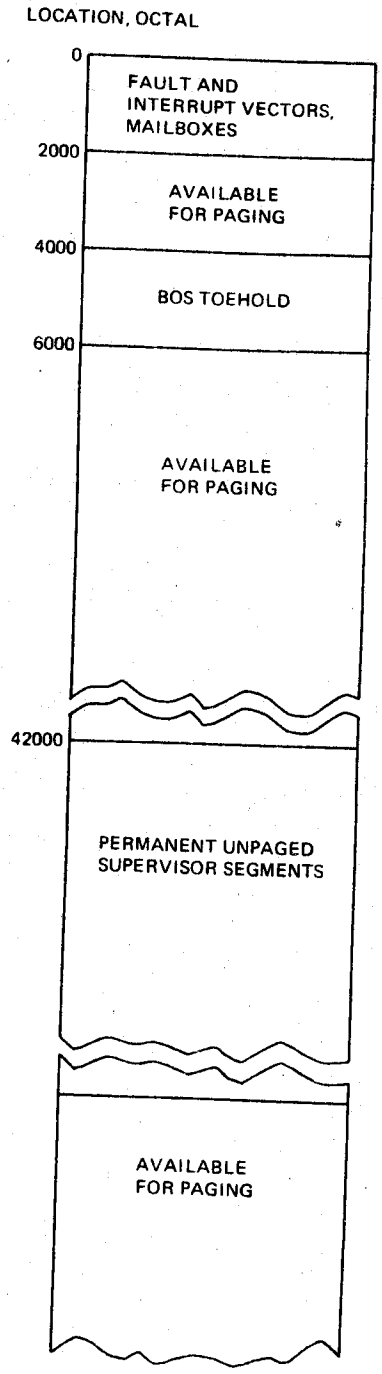

Figure 1-4

Main Memory After make_segs_paged

call oc_data_init	Initializes operator's console data bases. Reports to IOM manager for device index assignment.
call scas_init	system controller addressing segment (SCAS) set up. Processor and system controller configuration data processed and interpreted. RSW and RSCR instructions issued to verify this data.
call scs_init	interrupt assignments ascertained. System masks fabricated. Process interrupt handlers set up. Interrupt handler entry points set up for interrupt interceptor. Control processor relations set up.
call trace_init	debugging printer/tape facility set up, if selected. Reports to IOM manager for device assignment.
call init_sst	system segment table (SST) organized. Core and PD maps and used lists set up. AST entries created, threaded into free lists. Core left by bootstrap1 freed.
call initialize_faults$interrupt_init	interrupts directed to interrupt interceptor. Data collected by scs_init used to assign correct entry point.
call clock_init	local time zone ascertained from configuration deck, put in sys_info.
call initialize_dims	storage system I/O routines called to report to IOM manager. FSDCT accessed or constructed based upon INTK card. First page faults taken.

call make_segs_paged	new paged descriptor segment made. All segments other than permanent unpaged supervisor segments made paged. Main memory for initialization segments freed.
call syserr_log_init	segment made to access LOG partition of disk. Operator's console logging made operative.
call delete_segs$temp	collection 1 temp segs deleted.
call debug_check$copy_card	DEBG card interrogated - system debugging options set.
call tape_reader$init	initialize tape package, with respect to IOM manager. DCW lists to read tape into paged buffer segment set up.
call trace_rsw	check switches to return to BOS.
call segment_loader	load collection 2. Prelink it.
call init_str_seg	system trailer segment set up as list of free trailers.
call init_hardcore_gates	linkage pointers set in hardcore gates for performance optimization. Ring brackets set in SDWs for fault restart mechanism.
call build_template_dsegs	an obsolete data base is constructed.
call getuid$init	storage system unique ID generation initialized.
call init_sys_var	miscellaneous system variables initialized. Required error_table_ codes copied into wired data bases.
call init_root_dir	root constructed on cold boot. initialize_kst called to allow segment initiation. Root initiated.
call initialize_faults$fault_init_two	all fault handlers set up.

call init_branches	segments which would go into the hierarchy are put there. Branches constructed, and connected to AST entries. Branch attributes copied from SLT.
call delete_segs$temp	collection 2 temp segs deleted
call trace_rsw	check switches to return to BOS.
call load_system	load collection 3. Loaded into hierarchy, not prelinked.
call trace_rsw	check switches for return to BOS.
call tape_reader$final	MST rewound, tape_io ceases communication with IOM manager.
call tc_init	traffic controller set up. Initializer process made out of current state of system. Bootload idle process set up.
call io_init	typewriter package set up, sets up buffers and control words. I/O interfacer initializes self. DATANET 6600 FNP communication initialized.
call delete_segs$delete_segs_init	all initialization segments deleted. Current procedure (initializer) is a supervisor segment.
call init_proc$multics	system search rules set up. system_startup_ in ring 1 called.

SECTION II

COLLECTION 1

Collection 1 of the Multics System Tape contains all of the procedures and data bases necessary to make paging operative. Since all programs and data must fit in main memory before paging is operative, collection 1 contains the minimal number of segments necessary to attain this end.

Collection 1 must contain the programs and data necessary to take the minimal information passed by BOS, including the CONFIG deck, and construct a paged Multics environment. The proper routing of faults and interrupts and the determination of the system configuration are among the functions performed by these programs. The creation of the PL/I environment, e.g., stacks and symbolic references, is another.

All of the programs and data bases loaded as part of collection 1 are loaded by the program bootstrap1 (with the exception of bootstrap1 itself). All of these procedures and data bases are loaded into contiguous unpaged segments in main memory. After loading, a call to init_collections$init_collection_1 makes calls to many procedures in collection 1 to initialize many of the data bases and subsystems loaded. By the time this has been done, many of these segments are paged.

bootstrap1

bootstrap1 loads itself into main memory, sets up an appending environment, sets up segments describing hardware-defined areas and information passed by BOS, and loads the rest of collection 1.

The first tape record of bootstrap1 receives control from BOS in absolute mode at location 40 relative to the base of bootstrap1. Immediately, the processor maintenance panel switches are checked for a pattern to cause a return to BOS (see Section I, "The Environment Passed To Initialization"). Next, interrupts are masked. All code up to this point has been running with the inhibit bit set on, but this usage must be minimized if lockup faults are to be avoided. Thus, bootstrap1 determines which port on the bootload system controller the bootload processor is connected to by issuing a RSCR-CFG (read system controller registers, configuration) instruction, directed at the bootload controller. Among the information returned by this instruction is this port number. Knowing this number allows the SSCR-IER (set system controller registers, interrupt enable register) instruction to be issued, setting the interrupt mask assigned (via the EIMA [1] switches) to that port. Once this mask has been set, masking all interrupts, the inhibit bit is no longer used, preventing a lockup fault. The interrupt vector is now filled with NOP pairs, causing all interrupts to be ignored. The fault vector is set to ignore timer runout faults. The timer (processor timer register) is also loaded with a very large number. The interrupt mask is now set to enable all interrupts, as they will be effectively ignored.

bootstrap1 now obtains an IOM PCW left by BOS in the first words of the IOM mailbox to determine the tape channel and drive number of the Multics System Tape (MST). PCWs and IDCWs to be used by the tape-reading routine in the first record of bootstrap1 are appropriately initialized. A loop is now entered that reads in all of the remaining records of bootstrap1, using these PCWs and IDCWs. The length of bootstrap1, i.e., the number of tape records to be read, is determined from the SLT entry of

---

[1] EIMA stands for Execute Interrupt Mask Assignment. There are four EIMA switches on a Series 6000 system controller. Each switch is associated with a mask, which masks the system controller interrupt cells. Each switch selects a port on the controller. If an interrupt cell is set, and unmasked by the corresponding bit of a given mask, the port selected by the corresponding EIMA switch is sent a signal to create an interrupt.

bootstrap1, which, as you will recall from the last section, precedes bootstrap1 on the MST and is loaded as part of it.

When all of bootstrap1 is read in, the switches are checked for a return to BOS and a descriptor segment is set up at a fixed location relative to the beginning of bootstrap1. Descriptors are constructed for all of the hardware-defined areas of main memory (DATANET 6600 FNP mailbox, the IOM mailbox, the bulk store mailbox, fault and interrupt vectors, floating fault vector) and the configuration deck. The address of the IOM mailbox was supplied by BOS in an index register. All of these other areas are assumed to be at known locations. These segments are all permanent supervisor segments. The Segment Loading Table (SLT) and its Name Table are laid out as areas following the descriptor segment. Descriptors are constructed for them, and they too are permanent supervisor segments. The Physical Record Buffer is laid out following the SLT Name Table, and descriptors describing it and bootstrap1 are constructed, being the first two descriptors for initialization segments. The pointer registers are loaded with pointers to the base of several of these segments. A transfer instruction is executed whose final address was prepared via appending, causing appending mode to be entered.

The interrupt vector is now filled with SCU-TRA pairs, ignoring all interrupts but causing machine conditions to be stored. The fault vector is set to cause a fatal error if any fault is encountered except timer runout, which is ignored. These SCU-TRA pairs use ITS pointers to prepare their final addresses. These ITS pointers are set up following the fault and interrupt vectors. The actual SCU and TRA instructions never change from this point on. Only the ITS pointers are changed. The segment loading table is initialized with a template describing all segments set up so far. A more powerful tape-reading routine, which is part of bootstrap1, is now initialized. This initialization includes priming its buffer with the next tape record. The amount of available main memory is ascertained from the configuration deck MEM cards. The data switches are again checked for a possible return to BOS.

Collection 1 is now read in. Preceding each segment is the SLT entry for this segment, which is itself preceded by a header word giving its length. The SLT entry is loaded into a standard location and the names and possible ACLs and directory pathname that follow the SLT entry are copied into the name table. A segment number is allocated to the segment being loaded, with the next sequentially available segment number, starting at zero for supervisor segments and 400 for initialization segments. Checks are made that the segment being loaded follows in the proper sequence of text and linkage segments, i.e., that each text

segment with the link_provided bit on is followed by exactly one linkage segment. The first name of the segment is checked against a table known as interesting_names, which directs special-case action to be taken for certain segments. This action consists of either remembering the segment number of the segment being loaded for the transfer to bootstrap2, or setting the length of the segment based upon the configuration deck (e.g., zeroing the length of d190_seg if no DSS190 subsystem is configured and setting the length of tty_buf from the TTYB card).

Space is allocated for the new segment. This space is the next available space in main memory of the length specified by the possibly-adjusted bit count, either starting at the low end of memory for unpaged supervisor segments, or starting at the high end for all others. (See Section I, "Main Memory Management").

Next, a SDW is created for the segment, using the bit count in the SLT entry (which may have been modified by the special-casing). This SDW allows read and write access for the purpose of loading. The adjusted SLT entry is copied into the correct place in the SLT and the number of words specified by the MST header word are read into the new segment. (This header word precedes the text of the segment and tells how many words are written on the tape as opposed to the bit count, which tells how much space should be allocated.) Finally, the access bits specified in the SLT entry, including the encacheability control bit, are placed in the SDW.

A special MST header word signals the end of collection 1. At the time this is encountered, bootstrap1 computes the amount of main memory remaining between the supervisor segments and the temporarily-unpaged segments and places its address and length in the SLT header. The data switches are again checked for a possible return to BOS. Pointer register seven (SB) is made to point to the base of the segment pds, whose segment number was remembered during the special-casing in loading. The timer register is reloaded with a very large number and the processor tag of the bootload CPU is ascertained with an RSW instruction (which must be done by a privileged procedure). The processor tag and the segment numbers of the SLT and the SLT manager (the latter's segment number was also remembered during the special casing) are loaded into index registers. A transfer is made to the first word of the procedure bootstrap2, whose segment number was similarly remembered.

At this time, the fault and interrupt handlers of bootstrap1 are still referenced by the fault vector, but are not used except in case of error.

bootstrap2 AND PRELINKING

bootstrap2 is the program transferred to by bootstrap1. It is loaded by bootstrap1 with the rest of collection 1. It is responsible for setting up the PL/I environment, i.e., setting up enough mechanisms so that PL/I programs can be used. The principal features of this environment are stacks, stack segments (with various pointers in their bases, see the Runtime Environment PLM, Order No. AN84, for information on the PL/I environment), and symbolic references.

bootstrap2 begins by initializing the linkage pointer register (LP) to the base of its own linkage section. This is possible without SLT searching because the linkage section of bootstrap2 follows bootstrap2 on the MST and both are initialization segments and hence, their segment numbers are contiguous. This allows bootstrap2 to make symbolic references after prelinking has occurred. Next, a stack frame is set up. This involves the initialization of stack begin and end pointers in the PDS, used as the stack (recall that pointer register 7 points to the base of the PDS at this time) and the in-line execution of a standard Multics push macro. This having been done, a stack frame is available for use and others can execute push and return macros when called by bootstrap.

The program slt_manager is used by prelinking and the remainder of initialization to build SLT entries and scan the SLT for segment names. It must be initialized next by passing it the segment number of the SLT, which was passed by bootstrap1. It then scans the SLT (which contains a pointer to the name table) for the segments lot and lot_maintainer, which it uses during prelinking. All of these segment numbers are stored in impure storage in the procedure slt_manager. The call to slt_manager is made via a transfer vector at the beginning of that program. All calls made between bootstrap2, slt_manager, lot_maintainer, pre_link_1 and pre_link_2, until prelinking is complete, are made via such transfer vectors at their respective beginnings as links for symbolic references have not been resolved.

Once the initialization of the SLT manager is complete, bootstrap2 calls the SLT manager (again, of course, via the transfer vector at the latter's beginning) to ascertain the segment numbers of pre_link_1 and pre_link_2. pre_link_1 is now called, with the segment number of the SLT manager as an argument. pre_link_1 is responsible for scanning the entire SLT to locate all links that can be snapped at this time. It is also called after the loading of collection 2 to snap all links that can be snapped at that time. pre_link_1 begins by calling the SLT manager to obtain the segment numbers of the LOT maintainer,

the LOT segment, the hardcore combined linkage segments (active_sup_linkage and wired_sup_linkage), the descriptor segment, the SLT, pre_link_2 (the second half of the prelinker), and the initialization combined linkage segments (active_init_linkage and wired_init_linkage). pre_link_1 scans the SLT entries for all of the supervisor segments loaded so far (segment numbers begin at zero) and then all of the initialization segments loaded so far (segment numbers begin at 400). The highest and lowest segment numbers in each category are determined from the SLT header. The SLT entry of each segment is inspected to see if the link_provided bit is set. This bit is turned on by the use of the linkage keyword in the MST header file. It indicates that a separate linkage segment is provided for this segment, whose segment number is in the segno field of the SLT entry. If there is a linkage segment provided, the combine_link bit is inspected to see if that linkage section should be combined into an appropriate combined linkage segment or left self-standing.

The choice of an appropriate combined linkage segment into which to combine a linkage section is based upon whether the owning segment is a supervisor or initialization segment and whether or not the link_sect_wired bit in the SLT entry for the segment is on. The appropriate selection of active_sup_linkage, wired_sup_linkage, active_init_linkage, and wired_init_linkage is made from these considerations. If the linkage section is to be combined, it is combined in the appropriate linkage segment and a pointer to that section of the combined linkage segment is placed in the LOT entry (in the segment lot, which is the linkage offset table for the supervisor) for the segment whose linkage is being processed. If the linkage is not to be combined, a pointer to the uncombined linkage section is placed in the LOT. A bit (slte.pre_linked) is set in the SLTE of the segment so that pre_link_1, when called again after the loading of collection 2, does not reattempt to combine the linkage for this segment. After the LOT entry has been set, the appropriate pointers at the head of the linkage section are initialized, just as in normal linking. The procedure lot_maintainer, which is used by the normal system linker, is used for many of these functions, but it is called via transfer vectors at its beginning for these uses.

pre_link_1 causes all of the LOT entries for procedures and data bases loaded in collection 1 to be set and all of the linkage of collection 1 that must be combined to be combined. After this has been done, the linkage sections of all of these procedures and data bases are scanned for unsnapped links. (The linkage sections of data bases should not have links, but are useful insofar as they are set to contain pointers to the definition sections of their owning segments. Use of the

pseudo-op movdef in the assembly of these data bases normally causes the actual definition section to reside in the linkage section as well to avoid its being overwritten when the data base is actually used.) During collection 1 prelinking, all links are found to be unsnapped. During collection 2 prelinking, after collection 2 has been loaded, all links that were snapped during collection 1 prelinking will be found to be already snapped. pre_link_2$force_link is called to snap each link. This procedure is responsible for locating the target of each link, using the SLT manager to convert an external name into a segment number for most links and scanning definition sections of text segments for inbound symbol names. This procedure contains code to search definition sections for symbols, identical to that in get_defptr (see get_defptr in the Binding, Linking, and Makespace Management PLM, Order No. AN81 for details on definition sections) used by the standard system. linker. Links that cannot be snapped at this time, i.e., which either reference a segment not loaded yet or in error, are left unsnapped. Such links are enumerated by the MST Checker Program (see check_mst in the System Tools PLM, Order No. AN51) as "not found." It is legal and legitimate for programs in collection 1 to reference programs and data bases in collection 2. Only the actual execution of such referencing code before collection 2 prelinking is illegal. Attempts to make such references cause linkage faults, which are directed to the unexpected fault entry of the interrupt interceptor, causing system failure.

After the prelinking of collection 1 is complete, control is returned to bootstrap2. bootstrap2 can now make symbolic references and proceeds to do so at once. The bootload CPU tag, passed by bootstrap1 in an index register, is stored at scs$bootload_cpu_tag, in the system communication segment (SCS). This information is needed at reconfiguration time. The pointers to the required operators in the pl1_operators_ operator segment are stored in the correct places in the stack headers of the PDS, which is currently being used as a stack, and the PRDS (segment prds), which is used as a stack at interrupt, page fault, and scheduling times. A pointer to the linkage offset table (LOT) of the supervisor, segment lot, is placed in the appropriate place in both of these stack headers. A pointer that would cause a fatal fault (out of segment bounds on descriptor segment) is placed in the position in both stack headers where a pointer to a signalling procedure is expected. This causes system failure if any attempt is made to signal any condition before the fault mechanism is fully initialized in collection 2 initialization. The SDW for bootstrap1 is also zeroed at this time as a matter of cleanliness.

Finally, a call is made to initializer, a PL/I program that is a permanent supervisor segment. This procedure, the first PL/I program that runs, dispatches further calls for initialization. One of its last actions is to delete all of the initialization segments. Hence, it must be a supervisor segment. Initializer calls init_collections$init_collection_1 to initialize the data bases and subsystems loaded in collection 1. Initializer then calls segment_loader to load and prelink collection 2 and init_collections$init_collection_2 to initialize the data bases and subsystems in collection 2. Finally, a call is made to exit the supervisor environment into the administrative ring.

The loading of collection 3, a fairly easy task, is one of the functions performed by collection 2.

The rest of this section is devoted to the initializations of the various subsystems and data bases loaded in collection 1.

COLLECTION 1 FAULT INITIALIZATION

At this point in collection 1 initialization, the fault and interrupt vectors still point into bootstrap1, whose SDW was destroyed just a moment ago. There are no legal faults or interrupts at this time. The first call made by init_collections$init_collection_1 is to initialize_faults$fault_init_one to set up another temporary fault-handling policy and do some permanent initialization of the fault-handling mechanism.

First, the read switches (RSW) instruction is issued to ascertain which port on the bootload CPU is connected to the bootload system controller. From this information, a word that produces an effective address (internal segment offset) whose top three bits are this port number, is placed in sys_info$clock_ and prds$proc_contr_ptr. This allows the clock in the bootload system controller to be read at any time after this. The clock is ready by issuing a RCCL (read calendar clock) instruction, which indirects through sys_info$clock_. This measure is done now so that syserr, the operator's console error logging routine, can print out the time of day in its messages before the full CPU port-system controller port correspondence is determined. prds$proc_contr_ptr is set so that wire_stack, which is called in case of a syserr call, can set system controller masks. The value of these masks before they are initialized are the assembled-in values in the segment scs, a fully open channel mask and a fully closed interrupt mask.

Next, all interrupt ITS pairs in the fault vector segment are filled in, directing all interrupts to wired_fim$ignore, which summarily does an RCU when such an interrupt happens. The SCU data is directed at prds$ignore data. Note that the actual SCU and TRA instructions created by bootstrap1 are not changed; only the ITS pairs they reference are modified. All fault ITS pairs are now set to route all faults to ii$unexp_fault, storing SCU data at prds$sys_trouble data. Once all fault ITS pairs are set to this, lockup and timer-runout faults are redirected to wired_fim$ignore; timer runouts are meaningless until traffic control is initialized after collection 3 has been loaded. The mechanism to signal faults such as lockup is not yet ready. Connect faults are directed to wired_fim$connect_handler at this time. They are set up now since connect faults are part of the paging mechanism, which is initialized in collection 1 (although connect faults should not be received until other processors are configured in.)

Page faults (directed fault 1) are directed to page_fault$fault, their final destination, as there is paging before collection 2 is loaded. Segment faults (directed fault 0) are also directed to their final destination, fim$primary_fault_entry, as there are segment faults in collection 2 initialization before the remainder of the fault mechanism is initialized in collection 2. (Segment faults occur as soon as segments can be initiated. See "Root Directory Initialization" in Section III.)

Next, several programs (ii, fim, wired_fim, emergency_shutdown, restart_fault, page_fault, return_to_ring_0_) are modified by storing selected pointers in their texts. This is done by temporarily changing the access in the SDWs of these segments to allow such writing, storing the necessary pointers, and then restoring the access. The reasons for storing such pointers are several. Generally, these programs receive control from fault and interrupt vector TRA instructions. To save the pointer registers, these programs cannot use instructions that involve pointer registers in their address preparation as the contents of the pointer registers are unknown, and the pointer registers cannot be loaded until saved. Hence, the only legal type of address preparation is that involving operands and indirect words relative to the text segment (instruction bit 29 = 0) and ITS pointers. Since these procedures must be shared, pointer registers and other data cannot be stored in their text. Hence, at initialization time we store ITS pointers to the locations where pointer registers are to be saved in the texts

ITS pointers to allow the ordinary (A, Q, index, etc.) registers to be stored in the same way are also set up.[1] These procedures also frequently restore machine conditions stored at fault or interrupt time. In such sequences, the last instruction must be a Restore Control Unit (RCU) instruction. Again, since the pointer registers contain unknown quantities (viz., their contents at the time of the fault or interrupt) immediately before restoration of the control unit, text-imbedded pointers must be used for the address preparation of the RCU instruction. Also, these procedures ascertain their own settings of the linkage pointer (LP) register from text-imbedded pointers. These procedures cannot use the LOT to determine their linkage pointers as they have no way of locating the LOT, as no stack can be located until the correct stack segment (pds, prds) can be found. They find the stack segment via links, requiring LP to be set. The procedure return_to_ring_0_ also requires a text-imbedded pointer, this time to the first word of restart_fault (the latter is a gate segment, with a call limiter of 1). These programs are used by the fault-restarting mechanism (see the Process and Processor Control PLM for information on the fault-restarting mechanism), which can operate in the user ring and provide an orderly return into ring 0. This is discussed in detail in "Hardcore and Outer Ring Segment Numbers" in Section III. Notice that the SDWs describing both of these procedures (return_to_ring_0_ and restart_fault_) by their hardcore segment numbers must also allow access in the user ring.

HARDCORE I/O AND OPERATOR CONSOLE INITIALIZATION

Once the fault mechanism has been initialized for collection 1, allowing at least returns to BOS that can be analyzed in case of difficulty, the next area to be initialized is hardcore I/O. Hardcore I/O must be initialized so that the operator's console can be made operative as early as possible, so

---

[1]These ITS pointers, like the ones referenced by the fault and interrupt vectors, point to locations in the pds and prds reserved for the saving and restoring of SCU data. Although these locations have the same address (segment number and offset) in all processes and to all processors, the binding of this segment number is to a unique per-process segment (pds), or a unique per-processor segment (prds). Thus, the actual storage location referenced as this "shared" address is process or processor-dependent.

that informative error messages can be printed out in case of difficulty. Furthermore, the general aim of collection 1 is to make paging operative and I/O for the storage system is necessary for this end.

Initialization of the IOM manager and its IOM mailbox is the first and most important step of hardcore initialization. The device table array is zeroed. All per-IOM pointers are set to null until the per-IOM initialization, which is to follow. The absolute address of the IOM mailbox segment is stored in the IOM data segment (iom_data). The IOM stop LPW and stop DCW (see the Supervisor Input/Output PLM for information on the use of LPW and DCW control words) are set up. A loop is now entered for all configured IOMs doing a per-IOM initialization. The IOM tag, mailbox, and status queue pointers (special and system-fault) in the IOM data segment are set up for each IOM. Locks, IMWs, PCWs and other miscellaneous quantities are zeroed. The mailbox of every channel is initialized with zero SCWs, and zero DCWs. The "stop LPW" (stopping channel operation if used) is put in the LPW of the mailbox of each channel. The pointer to the interrupt handler for each channel is initialized to null. The connect operand word (COW) for each IOM is set up. A connect operand word contains the port number (on the system controller containing the COW) of the device to be connected. This number is ascertained from the CONFIG deck. The connect channel LPW and DCW are set up. The LPW for the connect channel for each IOM is set up. The LPW is set to point to a single PCW pair.

The two overhead channels of interest for each IOM (system fault and special status) are next to be initialized. Nonexistent "devices" supposedly using these channels are assigned device indices by calls to iom_manager$iom_assign. This allows the proper entries in the IOM manager to be assigned as interrupt handlers for these channels. DCWs are set up for the special and system-fault status queues, and are stored in the DCW and SCW slots in the mailboxes for these channels, which do not use SCWs. LPWs are set up to replenish the DCW from the SCW slot, with a "no-change" bit, so that the same DCW is continually refetched, providing circular status queues.

Any error occuring during IOM initializatin causes a return to BOS without a message. The operator console has not yet been initialized. It is the next mechanism to be set up.

Information about the nature of the operator's console (ASCII or BCD) and its channel number and IOM number are ascertained from the PRPH OPC CONFIG card. Defaults are supplied if there is no such card. The configured partitioning of the wired log buffer into syserr and dim call regions are also

determined and the segment oc_data is laid out correspondingly. Free area pointers used to allocate buffers in these regions are initialized to point to their respective beginnings. Clock times in the syserr data area are initialized. The IOM manager is called to assign a device index to the operator's console and set up ocdcm_$int_handler as the handler for operator's console interrupts. The absolute address of the syserr data region is stored in the data segment, for use in fabricating DCWs. The necessary DCWs for turning on the audible alarm, reading and writing data, and causing carriage returns on the operator's console are set up. A BCD carriage return escape sequence is also set up. If the type of the operator's console requires it, a prefabricated string is output to it at this time to set the tabs on this console. The operator's console can now be considered operative, although the logging mechanism is not yet set up.

At this point I/O assignments can be made for hardcore DIMs and operator messages can be typed out. The DIMs of the storage system have not yet made their assignment calls. This is done at the time page control is initialized.

CONFIGURATION INITIALIZATION

Although some I/O device control routines (DIMs) have reported to the IOM manager at this time and set up interrupt handlers, all interrupts are still being ignored. To utilize interrupts, the relation between CPUs and system controllers and the various port assignments must be determined. This determination allows masks and port addressing words (words that may be indirected through to direct commands to given system controllers) to be constructed. Furthermore, the amount of available memory must be determined, and its on/off status, to allow page control to be initialized. Variables used by reconfiguration must be set to reflect the initial configuration of the system. The configuration specified by the configuration deck must be verified to the fullest extent possible by interrogation of processors and system controllers with the RSW and RSCR instructions, respectively. It is the goal of configuration initialization, performed by scas_init and scs_init, the next two procedures called by initialize_collections$init_collection_1, to perform all of configuration initialization..

SCAS stands for system controller addressing segment. To motivate its need, we digress briefly for a discussion of system controller addressing. Most instructions encountered by a Multics CPU refer to data in main memory. The appending process

converts a segment number and a word number into an absolute address if no fault is taken in address preparation. This absolute address refers to some location in main memory. As main memory is distributed among the configured system controllers, the CPU port logic (similar logic exists in other active modules, e.g., the IOM) must decide which system controller contains the data being referenced. This decision is made based upon the absolute address computed and the port assignment switches on the processor maintenance panel. A data request is then directed to the appropriate system controller. A Multics system controller, however, has other functions besides its principal function as an interface to main memory. Interrupt cells, masks to mask these cells, and execute interrupt mask assignment (EIMA) switches to direct the interrupts set in these cells to processors also reside in the system controller. Furthermore, a controller serves as a routing station for connect signals sent by a CPU issuing a connect (CIOC) instruction to other active modules. A system controller can also be asked for clock time or to set or report the contents of many of its internal registers, including its configuration switches. The CPU instructions that cause these specialized commands to be issued to a system controller (RSCR, SSCR, RCCL, SMIC, CIOC, RMCM, SMCM) must somehow specify to which system controller the executing processor is to direct the specialized command--that is, out of which processor port the specialized command is to be sent. For the set memory interrupt cell (SMIC), read calendar clock (RCCL), set mask (SMCM), and read mask (RMCM) instructions, the processor port logic inspects the top three bits of the effective address (internal segment offset) computed by the instruction and directs the specialized command out of the port selected by those bits. For the RSCR and SSCR (and also CIOC) instructions, however, the port out of which the specialized command is to be directed is selected based upon the absolute address computed by the instruction, as for most other instructions. Hence, to use these instructions, which in their most general form request configuration information from a system controller, a technique must be present for creating an absolute address lying within the memory contained within a given system controller. Multics provides a paged segment, the SCAS, whose nth page (starting at zero) has an absolute address within the memory commanded by the system controller on port n. The areas pointed to by the page table words (PTWs) of this segment are not dedicated to this segment. Other pages or segments can move in and out of these areas. As the SCAS is accessible only in ring 0 and no data is either stored or read through this segment, it is not a security problem.

The first task of scas_init is to construct the SCAS. A segment named scas is loaded as part of collection 1. It is unpaged and 64 words long. scas_init builds a page table in

number on the system active modules for use in clock reading. This is done only for the bootload system controller. As each configured system controller is processed, the next processor appearing in the configuration deck (on a CPU card) is assigned as its control processor (the first EIMA switch of the controller must point at that processor). The system controller port to which this processor is connected (must be the same for all controllers) is saved. A check is made that no more than one CONFIG card for the same CPU (same tag) has been supplied and a table giving controller port, indexed by CPU tag, is constructed. When there are more system controllers than CPUs (there must be more or the same number) each system controller processed after the last CPU has been processed is assigned the last processed CPU as a control processor (the actual configuration must agree with these assignments). A check is made that there are not more processors than system controllers.

INTERRUPT CONFIGURATION INITIALIZATION

When all of the system controller and processor data has been processed in this manner, interrupt configuration intialization is performed. Although some of this processing is performed before the CPU/system controller initialization, it is logically one step.

The idea of this interrupt initialization is to construct several quantities: the channel mask, which is set in system controllers, describing which of their ports are in use; the interrupt masks, which are masks set in the system controllers via SMCM instructions, allowing different interrupts at different times; and the array of interrupt handlers to be used by the interrupt interceptor (ii) at interrupt time. A fourth quantity constructed at this time is the simulate pattern, a quantity that, when set in the interrupt cells of the bootload system controller, causes all possible critical interrupts to happen. (See the Reconfiguration PLM for further information on this quantity.)

The configuration deck is scanned for all cards that describe active modules and those that can create interrupts (set interrupt cells in system controllers). As each device (IOM, Bulk Store, or CPU) that is connected to a system controller port is processed (the processing for CPUs is done by the loop described previously, that assigns control processors) a bit is set in the channel mask being constructed, indicating that port is in use.

Interrupt cell assignments, i.e., which interrupt cell in system controllers are used for what purpose, are described by configuration card parameters for devices that produce interrupts and by a special card (the INT card) for software interrupts.

There are currently six software-created interrupts:

1. Stop--used to force a process into the traffic controller to enter the "stopped" state.

2. Preempt--used to force a processor into the traffic controller to possibly give up its process.

3. Interprocess Signal (IPS)--used to force a process to respond to an interprocess signal (see the Process and Processor Control PLM for details on the use of IPS interrupts.

4. Processor Initialize--used to cause a processor being configured in to come to life.

5. System Trouble--used to force all processors into the interrupt interceptor, and all to stop, except the bootload processor, which returns to BOS.

6. Syserr Log--used to cause an arbitrary nonidle process to take an interrupt, to copy data from the wired syserr buffer into the paged log partition.

The nature of an interrupt cell specification on either the INT card or IOM, D355 or BULK cards, is that of a (possibly degenerate) three digit octal number. The second and third digits specify the interrupt cell associated with the specified interrupt. The first digit gives the "interrupt state", or masking level, associated with this interrupt. State zero means this interrupt can be taken no matter what level the system is masked at. State one means that it can be taken if the system is masked at level 1, 2, or 3. State 2 means that it can only be taken at level 2 or 3, state 3 means level 3 only. The four masks corresponding to level 0, 1, 2, and 3 are known as sys_level, page_level, swap_level, and open_level, respectively. Only the system-trouble interrupt may be taken at sys_level (it has state zero). All interrupts may be taken at open level. As each interrupt cell assignment is processed by scs_init, the state assignment of that interrupt is determined, stored in an array (interrupt_state) in the SCS, and used to turn on a bit in a per-state interrupt cell assignment word, which is used to construct the masks. As each interrupt cell assignment is processed, the corresponding element in the per-interrupt cell

array scs$int_hlrs (interrupt handlers) is set to a pointer to the entry point that is to be called by the interrupt interceptor upon occurence of this interrupt. Furthermore, an array of arbitrary arguments for interrupt cell (scs$argument) is set up. These arguments, set by scs_init now, are passed by the interrupt interceptor to the interrupt handler at the time of an interrupt and their meaning differs depending upon the nature of the interrupt. Patterns for sending to system controllers with a SMIC instruction, for generating the six software interrupts are constructed as the INT card is processed. This pattern is used when reconfiguration requires throwing an EIMA switch on the bootload system controller, and its use generates all possible hardware interrupts. See the Reconfiguration PLM for more detail on the use of this pattern.

When all interrupt cell assignments have been processed, the four masks are constructed from the per-interrupt-state information gleaned from the cell assignments. scs_init now returns.

After the return of scs_init, initialize_faults$interrupt_init is called. As this procedure modifies the ITS pointers for interrupt transfer vectors, its first step is to mask to sys_level, using the newly-constructed sys_level mask to mask out all interrupts except system trouble. The state information in scs$interrupt_state is now inspected. All interrupts declared as having state zero are sent to ii$paging_interrupt_entry, all others to ii$pageable_interrupt_entry. Machine conditions are set to be stored at prds$interrupt_data and pds$interrupt_data, respectively. Preempt and sys_trouble interrupts are directed from the interrupt vector directly to their appropriate handlers. The interrupt interceptor is avoided for two different reasons[1] for these two interrupts. Processor initialize interrupts are directed to wired_fim$ignore. These interrupts are redirected at the time processors are added, during reconfiguration or later initialization. Finally, per-processor information in segment prds (the bootload CPU processor data segment) is set up via a

---

[1] Preempt interrupts require per-process data to be saved, as the processor will usually give up the process as a result of this interrupt. Hence, preempt cannot be treated as a normal non-paged interrupt, which would set up a frame on the prds. sys_trouble should modify as little data as possible, and use as few mechanisms as possible, for the system may be partially non-functional at the time it is issued, and information for crash analysis must be left as it was when the interrupt was sent.

call to prds_init. This information includes pointers that can be indirected through for system-controller addressing (set up in the interim by initialize_fault$fault_init_1, as was described in Section I, for interim mask setting), processor tag information, and patterns to be used for sending connects to other processors. Only after the SCS has been initialized can this information be ascertained. Finally, initialize_faults$interrupt_init unmasks (i.e., sets the "open_level" mask in the bootload system controller), and returns.

One other minor detail performed during configuration intialization is the determination of local time zone and its difference from Greenwich Mean Time.

By the end of configuration initialization the interrupt mechanism is fully operative. Note that the interrupt mechanism was not necessary, however, for the printing of syserr (operator's console) messages, which could function without it.

The next step in initialization is the initialization of page control.

INITIALIZING PAGE CONTROL

Initializing page control is the last and most important step in collection 1 initialization. This initialization consists of three stages: setting up of the System Segment Table, containing most of the data bases used by page control; initializing the storage system device control routines and the accessing or creating of the FSDCT; and paging all segments that have to be paged. These three major functions are performed by the procedures init_sst, initialize_dims, and make_segs_paged, respectively. Other functions, performed along the way, will be described.

Setting up the System Segment Table (SST)

The SST contains several data bases:

1. The SST proper, consisting of meters, configuration information, one-of-a-kind counters and indicators, flags, and the like.

2. The CNT (counters), consisting of mainly meters, but also some counter-type information about core usage and a table keeping track of temporarily-wired procedures.

3. The Core Map (CMP), which is an array with an element for each page of configured (ON or OFF), core, describing the contents and status of that page. These entries (Core Map Entries, or CMEs) are threaded into a list maintained by the page replacement algorithm.

4. The Paging Device Map, which is an array with an element for each record of paging device, describing the contents and status of that page. These entries (Paging Device Map Entries, or PDMEs), are threaded into a list maintained by the Paging Device replacement algorithm.

5. The Paging Device Hash Table, which is used at segment activation and deactivation time to ascertain if a given disk record has a copy on the paging device, and if it does, its location.

6. The AST or Active Segment Table, consisting of Active Segment Table Entries (ASTEs) describing the status of each active segment. Part of each ASTE is its Page Table.

7. Page Tables--Although each ASTE contains a page table, the page tables are a somewhat independent (of the AST) data base. The page table words (PTWs), which make up the page tables, are maintained by page control and used by the hardware.

The initialization of the SST consists of organizing these various data bases. First, fixed constants in the SST proper are set up, e.g., the sizes of the different AST entries, the absolute address of the SST, etc. Space is allocated for the core map, based upon the system controller information in the SCS gathered by scas_init. If there is a PAGE configuration card, the existence of a paging device is inferred. The device ID of this paging device is transformed into a paging device ID, i.e., eight is added to it and the resulting "paging device id" saved in the SST. Space is laid out for the paging device map to describe all of the records specified as being used by the PAGE card. The pointer sst.pdmap, which points to the base of the paging device map array, is set to point to the origin of the Paging Device map array (the entry describing record zero) even if record zero is not currently being used. Hence, if record zero is not being used, this pointer points to some location below the actual paging device map and possibly below the actual origin of the SST. The Paging Device Map (PDMAP) is started on a page boundary in the SST as it is periodically written out to the paging device via page control primitives. As the first record(s) of the part of the paging device being used will

contain this map, the first PDMAP entry is never used and thus contains a small header describing the PDMAP and its location in the SST. This header, filled in now and again by initialization, can be accessed by the salvager and BOS to allow them to interpret the copy of the PDMAP saved on the paging device. The paging device hash table is laid out after the PDMAP. Its size is a function of the number of PDMAP entries. Information about the hash table is also put in the PDMAP header.

Following the PDMAP hash table, on the next eight-word boundary, is the AST. AST entries of the four sizes are set up, smallest first, and threaded into circular lists, one list for each size. The number of each is determined from the SST configuration card. The marker fields (the low six bits of each ASTE, which allow it to be distinguished from a PTW in a backwards search by page control) are set up at this time.

Following the setup of the AST, the system controller information in the SCS, gathered by scas_init, is scanned. All complete main memory page frames between the end of the permanent unpaged supervisor segments and the first temporarily unpaged (to-be-paged) segment (remember, all segments are still unpaged) are added to a list of such page frames that is the core used list of page control. These page frames are marked as free. The core map entries of all page frames in the bootload memory are marked as abs-usable (cannot be deconfigured), and all those corresponding to page frames occupied by unpaged segments as abs-wired, meaning that they cannot be moved.

Finally, all of the space between the end of the AST, after all ASTEs have been set up, and the end of the SST, if it contains any integral page frames, is added to the core used list. This is done by the procedure free_unused_pages. This space is marked as not abs-usable, indicating that it might be claimed for the SST at any time, even though this feature is not currently implemented.

One other minor initialization might be mentioned. The first eight words of the SST are set to all ones. Page and segment control frequently use pointers relative to the SST. In case of programming error, wild stores to very low locations in the SST are easily detected because of this initialization.

Initializing Storage System Devices and the FSDCT

The procedure initialize_dims is called after the SST has been set up. It is responsible for making paging operative.

The INTK (INTaKT <sic>) card left by BOS, describing which partition of disk is to be used by Multics and whether or not a file hierarchy exists, is interrogated. If the partition name is SALV, a switch is set in the SST indicating that the salvager, not Multics, is being bootloaded. In any case, the configuration cards for all disk storage subsystems are scanned, in order of device ID. The lowest ID device having a partition of the same name as that specified in the INTK card (usually MULT, if not the salvager) is defined to be the "master device." The extents and starting record numbers of these partitions are copied into arrays in the SST. A call is made to device_control$init (via the transfer vector page$init) for each storage system device (disks and bulk store) configured. These calls are dispatched to the initialization entry points of the various device control routines. These calls allow the latter routines to report to the IOM manager for device index assignment and interrupt handler recognition.

Once these device control routines are ready to operate, the File System Device Configuration Table (FSDCT) must be accessed if this is a warm bootload, or constructed if a cold bootload. The FSDCT contains a bit map of free records on all of the various storage system devices, a file map for the root directory, information about MULT partitions, a file map for itself, and information relating to the relative success of the last shutdown and the existence of a paging device. The FSDCT is accessible to BOS and the salvager. In either case (warm or cold) the first step in the use of the FSDCT is the creation of its page table. make_sdw, which allocates page tables for segments during initialization, is called to perform this task. A page table is allocated, using the max_length supplied in the SLT. This page table is not threaded into any AST list as it neither has a branch nor is to be deleted at any time during shutdown or initialization. The flags aste.gtpd (assuring that the FSDCT never goes on the paging device, for several reasons) and aste.dnzp ("don't null zero page", assuring that access to the FSDCT via PTWs is not turned off should a page of it become zero) are set on in the AST entry of the FSDCT. An SDW is created and inserted in the descriptor segment. We now consider separately the cases of warm and cold bootloads.

In the case of a warm boot, we must access the existent FSDCT on disk. The first page of the FSDCT is defined to reside on the first record of the MULT partition of the master device

(which, you will recall, is the lowest ID device having such a partition). It is guaranteed that the file map of the FSDCT resides within this first page. Hence, the device address of the first record of the master device MULT partition is inserted into the first PTW of the page table of the FSDCT. Now, the device addresses of all of the rest of the pages are copied from the FSDCT into the page table for the FSDCT. This first reference to the FSDCT, to extract its own file map, causes a page fault on the first page, which is properly resolved as the device address in the PTW has already been filled in. The information about the previous sizes of MULT partitions is checked with the configuration deck to make sure that they have not shrunk. A message is printed if the information about the state of the last shutdown indicates that the salvager has been run.

In the case of a cold boot, the FSDCT must be constructed. A fixed number of records, starting from the first record of the master device MULT (or SALV) partition are logically allocated to the FSDCT. These record addresses are placed in the PTWs of the FSDCT. The first page of the FSDCT is now faulted on, read in, zeroed, and certain constants initialized. Each device having a MULT (or SALV for the salvager) partition is processed in device ID order. The information about the partition size is put in the FSDCT being constructed. Constant information (per-device) is set up. The bit map tables are set to all zeroes. This indicates that all of the records in the device are in use. Next, via calls to the page control page freeing primitive (via a special entry that does not consider possible duplication of a page on the paging device) every record in that partition of the device is freed. Finally, the page table of the FSDCT is scanned and the device addresses of all pages in core are withdrawn (marked as in use) from the FSDCT itself. Pages not in use are marked as being zero as the disk records that were logically assigned to them earlier were never actually withdrawn from the FSDCT. Finally, the file map of the root is nulled out.

In either the warm or the cold case, the FSDCT is now wired, allowing its use. It is very important to note that in all paging up to this point, the only paged segment being referenced was the FSDCT. As it has the aste.dnzp switch on, no pages of it can ever be deposited (returned to the FSDCT as free). As all pages of it are logically assigned in the PTWs in advance, no pages of it are ever withdrawn. Hence, the FSDCT is never used for its normal function before it is wired and, hence, fatal page faults on it (with the page tables locked) cannot be taken.

Next, the pdmap_seg segment is set up. This segment provides a means whereby the PDMAP can be written out to the first records of the in-use portion of the paging device. An AST

entry (with page table) is created for this segment by make_sdw. It, like that of the FSDCT, is not threaded into any AST list. A descriptor (SDW) is created for it and placed in the descriptor segment. The aste.gtpd bit is turned on in its AST entry, inhibiting copies of this page from being made by the page-multilevel algorithm on the paging device. The device ID assigned to this segment is that of the device being used as a paging device. It is not the paging device ID discussed earlier.[1]

Device addresses are assigned to this segment at this time. They are the first sequential records on that device being used as a paging device. Hence, when this segment is written out, it is to the first records of the paging device device. At shutdown time, the paging device map is copied from the SST into this segment and every page of the segment forcibly written out. This updates the entire PDMAP to its residence on the device of the paging device. At other times, page control causes the pages of the SST corresponding to the PDMAP to be written directly to the device of the paging device. (See the Storage System PLM for information on paging and the paging device.) The salvager also uses the PDMAP segment to access a previously saved PDMAP, when flushing the paging device during a salvage.

At this time, the Paging Device Used List is set up. All of the allocated paging device map entries are threaded into the Paging Device Used List. The paging device records specified as being not in use on the PAGE CONFIG card are then deleted from the used list. If this is Multics being booted (as opposed to the salvager), information about the paging device is copied into the FSDCT. Finally, static variables in the SST describing the paging device are set up.

At this stage, page control is now fully operative. The FSDCT (which is now wired and usable) and the PDMAP segment are the only paged segments at this time. The next step in the initialization of page control is to make all segments paged that are stated in the SLT to be paged.

---

[1] The paging device ID is the device ID of the device being used as a paging device, or'ed with "1000"b.

The Making Paged of Segments

At this stage of initialization, all segments read in by bootstrap1 are in main memory in unpaged contiguous segments. Those unpaged for the duration of the system are in low memory. The others, including initialization segments, are in high memory. There is now sufficient mechanism to make these "temporarily unpaged" segments paged. This is done by the procedure make_segs_paged (formerly update_sst_pl1). The basic strategy of this conversion is to copy these segments into paged segments, replace the SDW for the old segment by that of the new segment, and free the memory used by the old segment.

make_segs_paged begins by creating the new (paged) descriptor segment it is ultimately to use. make_sdw is called to create the ASTE/page table for this new segment. It is not threaded into any AST list (the flag slte.ds in the SLT entry of the descriptor segment informs make_sdw of this fact). The abs_seg (see Appendix A for a discussion of abs_segs) ds_seg is used to address this new descriptor segment. The first page is wired. Next, the SLT is scanned for segments marked as being paged. They are not now paged, but will be made paged. (The FSDCT and the PDMAP seg are <u>not</u> marked as paged in the SLT.) The segment numbers of these segments are put in an array and sorted on the base address of their segments, i.e., their order in main memory. During this processing, the SDWs for all permanently unpaged segments are copied from the old (unpaged) descriptor segment (dseg$) into the new (paged) one (ds_seg$).

Once the array of segment numbers sorted by ascending memory address is complete, it is processed from the highest address to the lowest. An ASTE/page table is created for each segment by a call to make_sdw (or make_sdw$unthreaded for the PRDS). make_sdw previously determined (from the SLT) the appropriate list (see "Memory Management" in Section I) onto which to thread the ASTE, as well as the appropriate size of page table. The descriptor for this segment, with write access temporarily added, is placed in the descriptor segment slot (in the current descriptor segment) for the abs_seg dir_seg. It is also placed unmodified, in its correct position in the new descriptor segment (ds_seg). An assembler-coded utility, privileged_mode_ut$swap_sdw_in_use, is called to copy, word by word, the unpaged segment into dir_seg, place the SDW for the new segment in the current descriptor segment, and clear the associative memory of the processor.

It is critical that the same program that moves each segment also clear the associative memory. This is because the PDS, which is being used as a stack, is copied by these means. Assume

that some other program than the one used to move the segment cleared the associative memory. Then the first program would move the segment, which in the case of interest is the PDS, being used as a stack. It would then call the second program to clear the associative memory, with either one of these programs actually changing the SDW in the descriptor segment. The SDW for the old PDS, which has just been copied, remains in the associative memory. The call to the second program involves saving the return point of the first program. This is done on the old PDS as the associative memory has not yet been cleared. When the second program clears the associative memory, it finds that the return point of the first program was not saved on the new PDS. Hence, the same program must both move the segment and clear the associative memory.

When the new segment is in use the main memory occupied by the old unpaged segment is freed if it was contiguous to previous memory freed by the mechanism or the highest-located segment in memory. Hence, all of the temporarily-paged segments at the high end of memory are freed in sequence, starting at the high end. This freeing consists of adding the page frames used by these segments to the core used list as each frame becomes free in sequence. This process does not free memory used by bootstrap1, the SLT, and other segments of collection zero.

The wiring of paged segments is accomplished by make_sdw in 24.4 and later systems. The page table words of these segments are marked with the wired bit before the AST entry is handed back to make_segs_paged. This allows segments in collection 2 to be wired in the same manner. The wirings of the descriptor segment, the PDS, and bound_sss_wired_ are special-cased by make_sdw. Only the first page of the descriptor segment and the PDS are wired. That portion of bound_sss_wired_ that contains the pl1_operators_ segment is wired. This is determined from a special segdef in this segment. Wiring is done conditionally, controlled by the "wired" bit in the SLT entry of a segment. In pre-24.4 systems, wiring of paged segments was done by update_sst_pl1, the predecessor of make_segs_paged, and there were no wired segments in collection 2.

When all segments that were to be paged are paged, several special paged segments are set up. An ASTE for the root directory is allocated on the proper AST used list. (make_sdw cannot be used for this purpose. The root is not a segment that has an SLT entry). The file map of the root is copied out of the FSDCT into the page table in this AST entry. Flags are set in the ASTE of the root. The root cannot be deactivated, is not permitted to go on the paging device (for integrity reasons), has a terminal quota account, has very large quota, no quota

checking, resides on the master device, and has its file-modified switch set. Its current-length is set appropriately. The ASTE/page table for the shutdown stack is set up at this time. A small ASTE is allocated, and threaded out of its used list. All of its pages are marked as null not in core, but wired (i.e., if brought in, they remain in). It is set to be on the master device, have no quota checking, and marked as a hardcore segment.

When all of the above has been done, that is, all segments that are to be made paged (except the descriptor segment) are paged and the root and shutdown stack are set up, a program called collect_free_core is invoked. This program frees all of the core occupied by collection zero segments and any other core that was simply never freed (the page at location 2000 is such a page). This program is essentially a garbage collector. It walks through the descriptor segment, looking through all of the SDWs for supervisor segments that describe unpaged segments (all unpaged segments at this time must be supervisor segments) and inspects the core map entries for any page frame of core that contains one or more words of the unpaged segment. If the core map entry indicates that the entry is in the core used list, it is left alone. Otherwise, it is specially marked. At the end of this process, all core map entries for page frames that contain any words of paged or unpaged segments or are emtpy, but part of the pageable core pool, are either in the core used list or specially marked. Now, the page frames containing the BOS toehold and the page table for the SCAS are marked as they are in use but do not contain any words of any segment (although the page frame with the SCAS page table almost always contains other segments). With all marking complete, the core map array is scanned. Any page frame in a system controller currently configured (ON) whose core map entry was specially marked is unmarked and added to the core used list. This assures that all core that can be used is available.

The FSDCT is now updated to disk (for integrity purposes, to allow a better chance of a successful salvage should the system fail at any point from here on). privileged_mode_init$ldbr is now invoked to switch the processor onto the new (paged) descriptor segment. The address spaces described by the old and new descriptor segment are almost identical. Thus, the loading of the DBR does not effect a transfer of control, stack switch, or other erratic behavior. Finally, the page occupied by the old descriptor segment is freed. The initialization of page control is now complete.

FINAL INITIALIZATIONS OF COLLECTION 1

With page control fully operational, the operator's console logging mechanism can be initialized. The PART LOG card is located in the CONFIG deck and the device ID, location, and extent of the syserr log partition are determined from it. An ASTE/page table is allocated for the syserr_log segment. It is not threaded into any AST list. The device addresses in its page table words are set to the device addresses of the records of the LOG partition. Flags are set in the ASTE of the syserr log partition: it should not go on the paging device (integrity/reliability), has no quota checking (it is in no directory), and should not have any zero pages nulled (it is not in the MULT partition--should a page become zero, it could not be freed, as its pages are not represented in the FSDCT). An SDW for the syserr log is constructed and installed in the descriptor segment. The syserr log is inspected. If empty, it is initialized. Pointers in the wired syserr buffer are set up, describing the log partition. A flag is set declaring logging to be operative.

At this stage in collection 1 initialization, all of the temp segs in collection 1 are deleted, via traversal of the temp seg AST list (see Section I, "Memory Management"). These segments include those that initialized configuration, I/O, and page control. Finally, debug_check$copy_card is called. This procedure sets several system-wide debugging options from the DEBG CONFIG card. These debugging options are interrogated by various system programs and are used to help locate system problems.

RETROSPECT ON COLLECTION 1

The order of initializations for collection 1 is reviewed below.

Segments are loaded by bootstrap1. Prelinking and the setting up of a PL/I environment is accomplished next. Then PL/I programs are used almost exclusively. Interim fault and interrupt handlers are set up--only a very few faults are now legal. I/O handling is set up, and the operator's console is initialized. System configuration data is ascertained and data allowing system controllers to be addressed and used for paging are set up. Interrupts are directed to appropriate handlers once system controller and CPU data is available. Masks are set up. Page control data bases are initialized. Some early paged segments, viz., the FSDCT and the PDMAP segment, are set up. The FSDCT is accessed from the storage system. Segments are copied

into paged segments. The final main memory management policy is put into effect. Main memory occupied by collection zero segments is freed. The operator's console logging mechanism is made operative.

SECTION III

COLLECTIONS 2 AND 3

The loading and initialization of collection 1 accomplished all that was necessary to set up a paged environment. With this paged environment, there are no more space constraints restricting the loading of segments. Hence, collection 2 contains the rest of the hardcore supervisor. The segments in collection 2 are copied from the physical record buffer directly into paged segments. There are no unpaged segments in collection 2.

The major tasks of the initialization of collection 2 are the accessing of the storage system hierarchy, the placing of segments loaded by initialization in the storage system hierarchy, the loading of collection 3, and the setup of traffic control. Collection 3 constitutes all those parts of the user and system control environments necessary to perform a storage system reload, which can load anything else. The programs in collection 3 are copied directly into segments in the storage system hierarchy. There is nothing to be initialized. Hence, the loading of collection 3 is part of the initialization of collection 2. The setup of traffic control involves the creation of the bootload idle process and the setup of the full wait/notify mechanism. The initialization of traffic control is left until last.

At this stage in initialization (collection 1 loaded and initialized), paging is fully operative. The system is running in a paged, segmented environment. Descriptors for all existing segments that were loaded from the MST or created are in place in the descriptor segment. The root directory has an AST entry but no descriptor. There is no concept of process and only one processor is running. The MST is positioned ready to read

collection 2. No segments are "known" (have KST entries, as the KST of any process has not been initialized) and segment faults cannot be taken. Nothing in the storage system hierarchy is known to exist other than the root directory.

LOADING OF COLLECTION 2

After the loading and initializing of collection 1 is complete, initialize_collections$init_collection_1 returns to the program initializer. initializer then calls tape_reader$init to set up the collection 2 tape reader (which will be described shortly). A check is made for the correct pattern in the processor maintenance panel data switches for a conditional return to BOS. Then, the program segment_loader is invoked to load collection 2.

segment_loader is a program knowledgeable about the format of a Multics system tape, i.e., header words, collection marks, SLT entries, etc. To read the MST, it calls tape_reader, which is knowledgeable about the format of a Multics Standard Tape, of which the Multics System Tape is one. tape_reader is knowledgeable about headers, trailers, administrative records, retry conventions, etc. To read the actual tape, tape_reader calls tape_io. tape_io is knowledgeable about tape controller commands, DCW lists, status, etc. To actually perform the I/O, tape_reader uses the IOM manager.

tape_reader$init, called by initializer, calls tape_io$init, which sets a static variable counting the number of reels encountered and returns. tape_io$init inspects the physical record buffer, the segment of the same name, and picks up the PCW used for reading the MST left there by bootstrap1. In it are the MST drive and channel numbers. Keep in mind that the physical record buffer, which was unpaged when it was used by bootstrap1, is now a paged, wired, segment, an init seg. tape_io$init sets up DCW lists for reading tape into the physical record buffer segment, paying attention to the fact that the buffer segment is not contiguous in main memory. Constant information in the header of the segment is set up. A call to iom_manager$assign_devx is made to assign a device index to the channel identified in the PCW as the MST channel for future calls to the IOM manager. This channel will be unassigned when initialization tape reading is complete.

segment_loader reads the MST header word for the SLT header for each segment on the tape and then reads the entire SLT header (SLT entry, names and ACLs) into an automatic array. slt_manager$build_entry is then called (via a standard PL/I call,

as opposed to the special techniques bootstrap2 used to call slt_manager) to place this entry into the SLT and SLT name table (which are now paged, permanent supervisor segments). Next, make_sdw uses the SLT entry of the segment (which has already been set up) to determine which AST list and what size AST entry are appropriate. The SDW returned by make_sdw is then placed in the appropriate place in the descriptor segment, with write access added, so that the segment can be read in. tape_reader is then called to read the segment header word (specifying how many words of the segment are actually on the tape), and that many words of tape are then read into the new segment. Finally, the correct access is placed in the SDW. segment_loader also checks, like bootstrap1, that the correct sequence of text segments and linkage segments appears on the tape. Each segment is loaded in this way. Segment numbers are assigned as in bootstrap1.

When the collection mark (a special type of MST header word) indicating the end of collection 2 is reached, pre_link_1 is called, to reprelink the system. The entire prelinking process, as described in Section II, is repeated. The only difference this time is that most links in the linkage sections of collection 1 segments will be found to already have been snapped. The SLT bit slte.pre_linked signals the prelinker not to attempt to recombine the already-deleted and prelinked linkage sections of collection 1. When the second prelinking is complete, another check of the switches is made for a conditional return to BOS. segment_loader then returns to initializer and init_collections$init_collection_2 is called to dispatch the calls for the initialization of collection 2.

PRELIMINARY COLLECTION 2 INITIALIZATIONS

Before segments of the storage system hierarchy can be accessed, via the normal segment-fault mechanism, several preliminary mechanisms must be set up. These include the AST trailer segment and many minor system variables. Hence, before the first segments are initiated, many small-order initializations are performed.

The first of these is the setup of the AST trailer segment str_seg (system trailer segment). This segment is used to store lists associating active segments with all of the SDWs of various processes which might describe them (see the Storage System PLM, for more details on the use of this segment). It is used at segment-fault time, deactivation time, termination time, and at certain times when a segment changes its encacheability state. The initialization of this segment consists of filling it with a list of free (trailer) entries. The SLT cur_length field (as

possibly modified by the TBLS card) is used to determine the length of this segment. The head of this list is saved in the SST (sst.tfreep).

Next, the storage system unique ID generator is initialized. This program generates unique binary identifiers for processes and storage system segments, based upon the clock time of bootload. Its starting value is initialized from the clock. Hardcore gates are initialized at this time. This initialization consists of storing linkage pointers into the texts of hardcore gates, similar to what is done for fault and interrupt handlers (see "Collection 1 Fault Initialization" in Section II). Hardcore gates (gates into ring 0) must ascertain their linkage pointers from text-stored pointers as they cannot use the segment number by which they were called to ascertain their linkage pointers via the LOT in their own ring. This is because the segment number by which they were called varies from process to process and these gates have no way of determining the segment number by which they were loaded at initialization time (their hardcore segment number) to enable them to use the LOT. The linkage pointers supplied in the outer ring cannot be trusted. (See the discussion of "Hardcore and Outer Ring Segment Numbers" later in this section). Another initialization performed by init_hardcore_gates (which stores the linkage pointers in the hardcore gates) is the setting of outer-ring accessible ring brackets in the SDWs for return_to_ring_0_ and restart_fault, the programs of the fault restarting mechanism. These ring brackets are set (from the ring brackets supplied in the SLT, the only time that SLT ring brackets go directly into a descriptor) so that these two programs can be used in the outer ring with their hardcore segment numbers. Signaller stores a pointer to return_to_ring_0_ as the return pointer in its stack frame, which was developed via a prelinked link, and return_to_ring_0 calls restart_fault via the text-imbedded pointer already described. See the Process and Processor Control PLM for a fuller description of this mechanism.

Next, many minor initializations are performed by the program init_sys_vars. The identification of the console to be used as the initializer's console and the name of the routine that will be used to attach it at the appropriate time are copied from the CONFIG deck (if supplied; otherwise defaults are used) into a system data base (active_all_rings_data). The Device Table in active_hardcore_data, describing the available storage system devices, is initialized from the SST. This data base is used by the procedure assign_device, at the time that segments are assigned devices when they are created by the storage system. The current clock time is stored in a system variable at the time of bootload. The error_table_ code for record quota overflow is

stored in the SST. This is necessary because page control needs the value of this code to signal record quota overflow errors, and error_table_ is not wired and, hence, may not be referenced by page control. The next segment number available for a supervisor segment is rounded up to the next zero mod eight number and stored in active_hardcore_data as the number of supervisor segments. This quantity is used to determine if a given segment number represents a hardcore segment number or an outer ring segment number (see the following discussion). It is also used as the first available user-ring segment number in process initialization. This number is also set in the segment pds as the stack base segment number for the initializer process (see Process and Processor Control, Order No. AN60) and in the descriptor for the descriptor segment, so that it is loaded into the DBR stack base field at the time traffic control is initialized.

HARDCORE AND OUTER RING SEGMENT NUMBERS

Before proceeding with the discussion of root and KST initialization, a consideration of hardcore segment numbers and multiple segment number assignments is in order.

A segment is an ordered array of bits that can be accessed via a segment descriptor word that describes that segment (either via a page table or directly). Hence, any number of segment descriptor words, possibly in the same or different descriptor segments, can describe the same segment. The paging mechanism provides a means for swapping pieces of segments into and out of main memory automatically. It does not affect the notion of segment. Any AST entry filled by any means describes a (paged) segment. Any descriptor that points at the page table within that AST entry allows access to that segment. Similarly, file maps in the storage system hierarchy describe segments as they can be converted into AST entries via the mechanism of activation. Any contiguous region of main memory is potentially a segment as a descriptor can be constructed to describe it. A segment number is simply an index, relative to the descriptor segment of a given process, for accessing an SDW. If more than one SDW describing the same segment appears in a descriptor segment, then more than one segment number can be used in that process to access that segment.

The segments loaded by initialization in collections 1 and 2 are all described by AST entries or are contiguous areas of main memory. In the initialization environment, there is a descriptor segment and precisely one SDW for each segment. These segments reference each other via their linkage segments (and in some

cases via pointers stored in their texts) where the prelinker has stored the segment numbers of referenced segments.

The segments that constitute the initialization environment at the end of initialization in fact constitute the hardcore supervisor. Now as Multics shares its supervisor among all processes and the supervisor references itself via segment numbers that are part of it at the end of initialization, it is clear that all processes must assign the same segment numbers to the supervisor segments. If there were no text-imbedded pointers and copies were made (per-process) of hardcore linkage sections (which would clearly have to be prelinked per-process and some of them wired, a clearly unreasonable overhead) this would not be so. However, due to the need for text-imbedded pointers, shared hardcore linkage sections, and pointers that must retain their meaning across processes, in general, hardcore segment numbers must be constant across processes. (Another reason for this constancy is that the code that switches processes must have the same segment number in all processes, or it would get lost as soon as it loaded the DBR).

Some of the segments that are part of the supervisor, i.e., loaded by initialization as permanent segments in collections 1 and 2, are also intended for use by outer-ring (nonsupervisor) programs. These segments include the gates into the supervisor and utilities (e.g., the pl1_operators_ segment, the area management package, and the signalling routines) used by the supervisor as well as by outer ring programs. The only way that outer-ring programs obtain access to segments is via the storage system hierarchy. A segment is initiated via a recursive mechanism (see the Storage System PLM). This consists of recursively locating the directory containing the segment, given its pathname, and searching that directory for the branch of the segment of interest. Any available segment number is assigned to the segment in the per-process Known Segment Table (KST) and information identifying the containing directory and the location of the branch in the containing directory stored in the KST. Later, when the segment number that was assigned is used in the process for the first time, a segment fault occurs, for there is no descriptor. The KST is inspected and the branch of the segment identified via the information left in the KST at initiation time is accessed. A field in the branch (entry.astep) tells if the segment is active (has an ASTE/page table). If not, the file map in the branch is converted into an ASTE via the process of activation. In either case, an SDW can now be constructed pointing at the page table in that ASTE and placed in the descriptor segment of the process. (If the segment was active, entry.astep locates its ASTE.) When an attempt is made to initiate a segment (make it known), the storage system unique

ID of the segment, found in the branch, is hashed into a hash table in the KST nf that process to determine if that segment is already known. If so, a new number is not allocated, no action is taken, and the existent number is returned. Thus, it is never possible to initiate a segment with more than one segment number in the same process.

The KST management algorithm starts assigning segment numbers at the number stored in active_hardcore_data$hcscnt (the number of supervisor segments rounded up mod eight in an initialization that has just been described). Actually, seven segment numbers above this are reserved for stacks (see the Process and Processor Control PLM). Hence, when a process is created, no segment number above active_hardcore_data$hcscnt corresponds to a segment as the KST is void (other than the first entry, which is set to describe the ring 0 stack (PDS) of the process). When segments are initiated (first by the supervisor init_proc on behalf of the outer rings, and later by outer ring programs) segment numbers are allocated to these segments via the KST allocation algorithm described. Whether or not the segment being initiated is part of the supervisor is not known at this time. All that is certain is that it has a branch in a directory and is thus capable of being initiated. If it had no branch (like most of the segments in the hardcore supervisor, which are described only by their AST entries, or are contiguous areas of main memory) there would be no way to identify it. Thus, all of those segments of the supervisor that are to be initiated by outer ring programs must have branches. At the time of the first use of the segment number allocated in the KST, the segment fault that results causes the branch of the segment to be inspected. The field entry.astep will be found at this time to indicate that the segment is indeed active and its AST entry is identified as that set up by make_sdw during initialization.

There will be, therefore, two descriptors for initiated supervisor segments in a process: one was placed there at the time the descriptor segment was created, and is a copy of that made by make_sdw during initialization. The other is created by the segment fault mechanism. The first of these SDWs has ring brackets of (0, 0, 0) and is intended for use only by the supervisor (it can only be used in ring 0). The access control bits in this descriptor are those specified in the SLT. The second descriptor has ring brackets and access control information derived from the branch. Hence, the segment can be referenced via two segment numbers in the process. We call the one created by initialization the hardcore segment number of the segment and the other the KST segment number. The segment is actually known by the KST segment number. The descriptor corresponding to the hardcore segment number (the hardcore

descriptor) is permanent--it cannot be modified by access control primitives, segment control primitives, initiation, or termination.

Now that the distinction between hardcore and KST segment numbers has been made clear, several special cases can be explained. Hardcore gates are initiated in user rings via the normal initiation mechanism. They, therefore, have two segment numbers in a process. When called, they execute in ring 0 due to the ring brackets in the KST associated descriptors for these gates. At the time they execute, they are running as the segment of their KST segment number in that process. They have no way to determine their hardcore segment number (an EPAQ instruction would return the KST segment number) and find their linkage sections via the ring 0 LOT. Hence, text-imbedded pointers provide the only technique for location of their linkage sections (text_relative code can be addressed without recourse to pointer registers).

The procedure signaller is a supervisor segment. It is invoked via the fault-handling mechanism in ring 0 when a signalable fault occurs. It saves the faulting machine conditions in its stack frame, which it has set up on whatever stack the fault occurred (in whatever ring), and calls (via a RTCD instruction) the per-ring signalling procedure in that ring to locate an appropriate condition handler. This call constitutes an outward transfer. signaller sets the return pointer in its own stack frame so that, when control is returned to that frame, control passes to the procedure return_to_ring_0_. This procedure will transfer control back into ring 0 via a call. Thus, when the handler returns to the frame of signaller, return_to_ring_0_ calls back to ring 0. return_to_ring_0_ calls (via a CALL6 instruction) a special gate, restart_fault, to validate and reload (possibly modified) the machine conditions. This pointer to return_to_ring_0_ is developed via a link in the linkage section of the procedure signaller. Hence, it uses the hardcore segment number of return_to_ring_0_ (which never has a KST segment number in any process, as it is not in the hierarchy). return_to_ring_0_ calls restart_fault via a text-imbedded pointer, again set up during collection 1 initialization and using a hardcore segment number. Hence, the hardcore descriptors for restart_fault and return_to_ring_0_ must be accessible in all rings. This modification of these SDWs is performed by init_hardcore_gates. This descriptor is later copied for the descriptor segment of each process This special-casing could have been avoided if the pointers to restart_fault and return_to_ring_0_ were stored in an accessible place in each ring. These pointers would have to be created by initiating these segments (which would then have to go in the

hierarchy) at an appropriate validation level as each ring is initialized. This was ruled out as too expensive.

ROOT DIRECTORY INITIALIZATION

At this point in initialization, the major task remaining with respect to the address space and the hierarchy is the initialization of the hierarchy (if a cold boot) and the connection of branches in the hierarchy to the ASTEs of user ring accessible supervisor segments. The most obvious way to access the necessary directories to perform these initializations is to initiate them in the normal manner. Presumably, abs-segs, forced activations, and other heavy-handed techniques could have been used, but there is no reason not to use the normal mechanism when it is available.

To initiate segments, a KST will have to be used. The KST currently described in the descriptor segment of initialization is an empty segment called (in the SLT) kst_seg. This KST will be the KST of the initializer process. The decision is made to use this KST to initiate the necessary directories during initialization. As this KST will become the KST of the process Initializer.SysDaemon.z, the segments initiated by initialization remain known to that process.

The next step in initialization is to make this KST usable. The program init_root_dir is responsible for these phases of initialization. First, initialize_kst is called with a special parameter. This procedure, initialize_kst, is used during process initialization. When called by init_proc for ordinary process initialization, it sets up the KST of the calling process (initializes hash tables, allocation areas, etc.) and sets the default search rules of the process. The special parameter passed at this time tells initialize_kst to perform only the first set of these per-process initializations; i.e., not attempt to set up the search rules at this time. The allocation areas and fixed constants in the KST are set up. The first seven segment numbers are reserved (via the normal segment number reservation mechanism) for the stacks of rings 0 through ring 6. The descriptor for the PDS (normally the PDS of whatever process is being initialized--in this case, the segment pds loaded in collection 1) is copied into the SDW slot in the descriptor segment for the ring 0 stack.

With the initialization of the KST (to be the KST of Initializer.SysDaemon.z) complete, segments can now be initiated if they exist. The root directory is known to exist. It was created by initialize_dims (its file map was filled with null

addresses) if it did not exist and it was activated by make_segs_paged. A call to makeknown$pname is performed, passing the name of the root (>) as a parameter. This primitive special-cases the initiation of the root. To make the root known, it is only necessary to allocate a KST entry (and segment number) as always and indicate (by placing the convention-defined unique ID of the root 777777777777, in the unique ID field of the KST entry allocated) that it is the KST entry of the root. The root is made known by its own reference name, >, so that future calls to initiate segments will find it. In a normal process, the root is made known by the first initiate call in the process, when recursive analysis of the pathname of the segment to be initiated finally leads to a search of the root for the correct ancestor of that segment. The root is always made known in the same way, i.e., placing a unique ID of 777777777777 in a KST entry. The only reason that the root is made known explicitly during system initialization is that this may be a cold boot and the root must be initialized as a legitimate directory and no other segments exist in the hierarchy.

It is possible to initiate the root simply by setting up such a KST entry because the unique ID in the KST entry is inspected by the segment fault handler, explicitly for this purpose, before an attempt is made to search the containing directory for the branch. If a segment fault is taken on a segment stated in the KST as having a unique ID of 777777777777, the variable sst.root_astep in the SST is interrogated to locate the ASTE of the root (which can never move). An SDW is constructed to point to the page table in this ASTE.

With the root now known in the initialization environment, it is now locked. Even if the root was void (just created), it can be locked as its lock word would be zero. In in either case, the active quota switch (which indicates that quota accounting is managed in the AST for this directory) in the root is turned on. The INTK CONFIG card is now inspected to determine if this is a cold or a warm boot. If warm, the root is unlocked. If a cold boot, the root is simply a segment with one bit turned on and a word (the lock word of a directory) nonzero. Thus, the normal initializations performed on a new directory are performed. The directory header and hash tables are set up. The root is given a quota equal to that set in its AST entry by make_segs_paged. Having been made into a normal directory, the root is now unlocked. init_root_dir returns to initialize_collections$init_collection_2.

At this time, segments can be initiated and created via the normal storage system primitives. These are used to create branches for those segments loaded by initialization that are to

At this time, segments can be initiated and created via the normal storage system primitives. These are used to create branches for those segments loaded by initialization that are to go in the hierarchy. This is the next step.

BRANCH CREATION AND CONNECTION

All the preceding mechanisms in collection 2 initialization have led us to the point where we can create and delete branches in the storage system hierarchy at will. We can now make the connections between branches in the storage system (which we will create if necessary) and the AST entries of segments loaded by initialization. This is the major task performed by init_branches, the next program called in the initialization of collection 2.

The first step in these hierarchy initializations is to delete the entire subtree rooted by >process_dir_dir, if indeed there is one, left over from the last bootload. The directories in this subhierarchy were the process directories of processes long since gone. The storage space occupied by this subhierarchy must be freed. Thus, a call is made to del_dir_tree, the normal system subhierarchy deleter, to delete the recursive descendants of >process_dir_dir. Next, a call is made to delete >process_dir_dir itself. The quota parameters of the root are saved before these calls and restored after them. The quota assigned to >process_dir_dir is set at bootload time and is a function of the number of processes allowed as specified by the TCD CONFIG card. It is set to this value at bootload time, and hence, is not drawn off the root. Thus, it is not returned to the root at the time this subtree is deleted.

Next, the four AST used lists corresponding to page table sizes of 4, 16, 64, and 256 words (as opposed to the temp_seg, init_seg, and hardcore lists) are traversed to find all those supervisor segments that must go in the hierarchy. The current occupants of these lists are all those supervisor segments that are to go in the hierarchy (they were threaded onto these lists by make_sdw, which determined this by the presence of slte.branch_required) and any directories that have been activated up to this point (including the root). There are also free entries on this list. Entries corresponding to anything except supervisor segments are skipped. The segment number is determined from the aste.strp field, which normally starts the trailer thread if this ASTE was created by make_sdw. (This field is then zeroed.) Otherwise, the segment number is determined from any trailers that this segment may have. This segment number is used in the determination of whether or not this is a

supervisor segment. This segment number is then used to access the SLT entry for this segment. init_branches$branch is then called to create a branch for this segment, based upon the pathname and ACL information provided in the SLT.

init_branches$branch is a utility that creates a branch from SLT information and initiates the (usually void) segment that results from that creation. init_branches$branch calls make_branches to create the actual branch.

make_branches attempts to append a branch of the specified name (the first SLT name) to the directory specified in the SLT pathname field. If the branch already exists, make_branches destroys it and tries again. If the parent directory does not exist, make_branches calls itself recursively to create that. In either case, once the branch is created, make_branches sets the ACL specified in the SLT for this segment in the branch and returns.

init_branches$branch then deletes the ACL term rw *.SysDaemon.* that the append call created if no ACL was specified in the SLT. The newly-created segment is then initiated and the max_length in the branch set (via the normal storage system primitive) to as many words as specified in the SLTE bit count. The pointer to the segment returned by the initiate call is returned.

Going back to the main loop of init_branches, which has just called init_branches$branch to create a branch for a supervisor segment, sum$getbranch_root_my is called to access the newly-created branch. This primitive, normally called at segment fault time, accepts a pointer to a segment, inspects the segment number to locate the KST entry of the segment, inspects the KST entry to locate the branch of the segment in its containing directory, locks that containing directory, and returns a pointer to the branch.

When the branch has been located, the segment is "activated" by placing the SST-relative pointer to the AST entry of the segment in the entry.astep of the branch. Thus, any process that takes a segment fault on this segment, accesses the AST entry created by initialization. The AST entry is then cross-linked (via its rep and par_ring fields) to the AST entry of its containing directory (which must have been active at the time the branch was touched, and will not have been deactivated since, since no segment faults were taken since then, and no other process is running). The AST entry of the segment is threaded into the inferior list of its containing directory. If it is the

first segment in its containing directory to be activated by this or any other means, the quota account of the containing directory is activated (i.e., set in the AST entry and marked in the directory as being there). The master-limit switch (indicating a descendent of >process_dir_dir) is set in the AST entry from the newly-created branch. The entry bound (call limiter) and entry bound switch are set in the new branch from the hardcore SDW, located via the segment number determined earlier from aste.strp. (This is the hardcore segment number for this segment. The segment number in the pointer returned by init_branches$branch is a KST segment number).

Finally, the maximum length of the segment is determined from slte.max_length (if given, otherwise from slte.cur_length) and a file map of the proper size is allocated in the directory for this branch, replacing the small one allocated by default when the branch was created. This is done now because bounds faults cannot be taken on these entry-hold-active segments, so the largest file map needed is set up now.

When all this has been done for a segment, updateb, the segment control primitive that updates a branch from an AST entry is called, which has the effect of copying the file map as maintained in the ASTE by page control up to this point into the directory. Also, it calculates the number of records actually used by this segment at this time, and places it in the branch. We will need this information shortly to update the quota accounts of the ancestors[1] of this segment. The directory containing the segment is then unlocked and quota account of the first ancestor of this segment, which has a terminal quota account, is updated. The segment is then terminated, removing it from the KST.

When this loop through the AST used lists is complete, all supervisor segments to be in the hierarchy are in the hierarchy, with their entry-hold switches on and hardcore SDWs pointing at them. Next, the maximum length in the branches for the SLT and its name table are set, via standard storage system calls, to their current lengths as determined by a storage system status call. This is necessary because their lengths at this time are different from their lengths specified in the SLT; they grew during initialization.

---

[1] An ancestor of a segment is either the containing directory of that segment, or the containing directory of the containing directory, and so on.

Next, >process_dir_dir is initialized. The name >pdd is added to this directory. Quota is set on this directory proportional to the maximum number of processes allowed as specified on the TCD configuration card. The standard allocation of quota for one process is moved to >process_dir_dir> zzzzzzzbBBBBBB, the process directory of Initializer.SysDaemon.z. This directory must exist as the segment pds was placed in it by make_branches. The ACL on >system_library_1 is set to sma for Initializer.SysDaemon.z, and status (s) for *.*.*. The ACLs on >process_dir_dir and >process_dir_dir> zzzzzzzbBBBBBB (the process directory of the Initializer) are set to sma for the Initializer. The name sl1 is added to >system_library_1. The directory >dumps is created if not already there, and given an ACL. The segment >online_salvager_output is created if not there, and given access rw to *.*.*. The ring brackets on this segment are (0,0,0).

The initialization of the hierarchy is now complete.

COLLECTION 2 WRAPUPS

Once the hierarchy has been initialized, most of the work of collection 2 is done. initialize_fault$fault_init_two is called. First the timer is loaded with a very large number (to allow changing the timer fault vector), and then all faults are directed to their normal handlers. The signalling procedure pointer in the base of the PDS is set to signal_$signal_, the normal procedure used for this purpose. The floating fault vector is set up and scs$faults_initiatized set to indicate that the full fault mechanism is initialized. Bounds faults can now be taken. (Remember that the vector for segments faults was set up during collection 1 initialization.) The search rules for Initializer.SysDaemon.z have not been set up yet, but the supervisor takes no linkage faults. This will be part of the process initialization of Initializer.SysDaemon.z.

Now, the temp segs of collection 2 are deleted, via a call to delete_segs$temp. The AST temp seg list is again traversed, and all of these segments truncated and destroyed.

COLLECTION 3

We are now ready to load collection 3. There is sufficient mechanism set up at this time to make this loading trivial. Collection 3 requires no SLT entries, no hardcore descriptors, and no special casing in the AST. Its segments are not even part of the supervisor, but must be in the hierarchy in order for

Multics to come to initializer command level. The data switches are checked for a conditional return to BOS. load_system is called to load collection 3. The data switches are then again checked for a conditional return to BOS.

load_system operates by reading each remaining SLT header on the MST into an automatic array. No SLT entry is made. init_branches$branch is called, as in collection 2, to create the necessary branch and directories, set the necessary ACL, and initiate the segment created. init_branches$branch is instructed, however, to provide write access for Initializer.SysDaemon.z if not already present. The MST header word for the segment is then read (tape_reader is used to read the MST) and as many words as specified therein are read in to the newly-created segment. If there was no write access for the Initializer specified in the SLT, it is taken back.

During this loop, the segments created are not terminated; they are touched in order to restore the SDW that may have been faulted by the revoking of write access. This allows BOS to patch these segments if the conditional return to BOS after loading collection 3 is invoked.

When the collection mark indicating the end of collection 3 is encountered, loading of segments stops.

All initialization of the supervisor is now complete except the initialization of traffic control and the initialization of user-accessible I/O. After this is performed, the initialization of the process of Initializer.SysDaemon.z will be performed, and initialization will be complete.

INITIALIZATION OF TRAFFIC CONTROL

When all other mechanisms of the supervisor are functional, the bootload idle process must be set up to occupy the bootload processor when it is not needed. The ability to share the available processor(s) among many processes and the ability to create these processes also must be set up. The initialization environment, as it stands now, must be transformed into the process of Initializer.SysDaemon.z. These are the goals of traffic control initialization.

The first task of traffic control initialization is to establish the usability of the segment tc_data, which contains the data bases of traffic control: the Active Process Table (APT), the Device Signal Table (DST), and the Interprocess Transmission Table (ITT). The sizes of these tables are copied

off the TCD CONFIG card. These tables are laid out in the segment tc_data, and pointers to their bases set in the header of tc_data. APT entries and ITT entries are threaded into free lists.

Next, the scheduling parameters (max eligible, min eligible, working set factor, etc.) are copied from the SCHD configuration card into an assigned location in tc_data, so that they can be used by the scheduler.

Next, the procedure build_template_pds is called. The output of this procedure is a supervisor segment called template_pds. This segment is copied by process creation (see the Process and Processor Control PLM), when processes are created in order to create their PDSs. It is set up by copying the current PDS (>pdd> zzzzzzzbBBBBBB>pds) from its base through the end of its first stack frame. Before this copying is done, however, the first stack frame on (our) PDS is modified to appear as through the instruction before the entry point init_proc$init_proc has made an external call. This is done so that when a new PDS is returned to by the traffic controller, the first time a new process[1] runs, init_proc$init_proc receives control. As the traffic controller assumes that pds$last_sp points to the last valid stack frame on a PDS to which it is returning, this location in the template PDS is set to point to the stack frame being fabricated. The segment number in all pointers constructed by build_template_pds is that of the current PDS. That is the segment number of the PDS of any process in that process. The stack_begin_pointer and stack_end_pointer pointers in the template PDS are set appropriately.

As hardcore address spaces for all processes are essentially the same, descriptor segments for new processes are copied from the descriptor segment of the initializer. Thus, all hardcore SDWs are the same and in the same position as the process of the initializer. SDWs for the KST, PDS, and descriptor segment of new processes, however, are placed in the positions in the descriptor segment of the new process corresponding to the positions occupied by the supervisor segments kst, pds, and dseg

---

[1]The traffic controller checks a flag when returning to a process to see if this is the first time that this process has ever been entered. If so, getwork (the dispatching routine) was never invoked in the new process, and thus cannot return to its caller. In this case, a return sequence is executed on the PDS, to a stack frame previously set up. This special exit will play an important role later on, when the Initializer becomes a process.

in the descriptor segment of initialization (and thus the descriptor segment of Initializer.SysDaemon.z). The KST, PDS, and Descriptor segment itself of initializer.SysDaemon.z are those of initialization. The descriptor corresponding to the hardcore segment prds, however, is another matter. This will be described more fully.

Once the ability to create processes has been set up, the next step is to convert the initialization environment into the process of the initializer, Initializer.SysDaemon.z. An APT entry is allocated. A process ID, defined as APT entry offset concatenated with 777777 of the initializer, is defined for the initializer and set in tc_data as tc_data$initializer_id. The pointer pds$apt_ptr in the PDS, to become the PDS of the initializer, is set to describe this APT entry. A lock ID is obtained from the storage system unique ID generator and stored in the PDS and the new APT entry as the lock ID of the initializer. The initializer is set to have a timax of zero, ensuring good response time for this crucial process. The AST entry offsets of the descriptor segment and PDS in use at this time and to become those of the initializer, are stored in the new APT entry. The DBR value for the initializer process is copied from the descriptor segment, being the descriptor for the descriptor segment itself. You will recall that the stack base segment number for the initializer process was stored in this descriptor in the field corresponding to the stack base field of the DBR as one of the early initializations of collection 2. Finally, the new APT entry is threaded into the now-void APT ready queue and set to describe a process that is eligible, loaded, ready (not running), and has never run. The APT ready queue has never been inspected and the traffic controller getwork routine (see Multiprogramming and Scheduling, Order No. AN73) has never run. The number of eligible processes in the system is set to 1.

A minor initialization is performed now: all of the polling time clocks are set to the current clock time. This forces all pollings to be done the first time these clocks are inspected.

The last and most important step in traffic control initialization is the creation of the bootload idle process and the starting of the bootload CPU.

Idle Processes

An idle process is a process that does no useful work. Except when interrupted by preempt or other interrupts, it is in the program init_processor, in ring 0, masked at open level, usually executing a DIS instruction. Like all other processes, it is very egotistical, considering only its own useless work important and all interrupts simply diversions. Some interrupts, i.e., preempt, cause the process to be suspended and resumed at a later time. An idle process exists for essentially no other reason than to be preempted. It is always eligible and loaded. It is always in the traffic controller ready queue, although in a special place. When a processor in getwork seeks work, there is always the useless work of the idle process to do if nothing else can be found. Its work can always be resumed and is infinite in extent. As a processor must have an idle process available at all times, there must be one per processor. They are created and destroyed (by start_cpu and stop_cpu, respectively) as processors are added and deleted, during initialization and reconfiguration. When a processor is started, it picks up the DBR value of its associated idle process and runs in that process. It may be preempted but, from the viewpoint of that idle process, it always comes back to run that process. Every time it proceeds from its DIS instruction (a return from an interrupt causes the next instruction to be returned to), it proceeds once more to execute this DIS. However, every time that this loop is restarted, reconfiguration flags are checked. There are two such flags. The first tells a processor to delete itself. Deletion consists of being sure that all interrupts directed at this processor are now directed at some other one, being sure that no interrupts are lost, indicating that we are deleting ourselves and stopping in a nonpreemptable loop. Once this has been done the idle process for the CPU is deleted (to be described later). The second flag tells a processor that it is now the control processor of some other System Controller than it had been until now--it should change the proc_contr_ptr pointers in the SCS and its own PRDS. Whenever either of these functions is necessary, the idle process of the CPU that performs these tasks is given top priority in the ready queue and given its processor via preemption. (See the Reconfiguration PLM, for more details about these reconfiguration operations.)

The creation of an idle process consists of creating its descriptor segment, its PDS, and the PRDS of the processor to be started, and sending the processor being started, into init_processor in this process. The deletion of an idle process consists of deleting the processor (as described) and destroying the PDS, descriptor segment, and PRDS associated with that processor and its idle process.

It can be seen at once that a processor and its idle process are intimately associated. Each, in some sense, belongs uniquely to the other. A processor starts in that process, returns to that process, and deletes itself in that process. That process is run only by that processor. Associated with the processor and its idle process is a PRDS, or processor data segment. Other than the per-processor data in the SCS, this is the major per-processor data base. Other than at the time it is created, it is accessible only to its native processor. When a (nonbootload) idle process is created, a PRDS is created for it (plm$makeseg performs these creations). It is initialized by prds_init, just as the bootload processor PRDS was during initialize_collections$interrupt_init. An SDW describing it is put in the descriptor segment being built for the idle process in the position corresponding to the supervisor segment prds loaded during initialization in the initialization descriptor segment. (When the idle process of the bootload CPU is created, this descriptor is copied from the descriptor segment of initialization. Hence, in each idle process, from the time it starts, the segment number corresponding to the SLT name prds references the PRDS native to that processor and idle process. What is more, every time a processor switches processes in the traffic controller it carries the SDW for its PRDS with it in the AQ register and places it in the same position in the descriptor segment of the new process. Thus, for any processor, no matter what process it is in, the segment number assigned in the SLT to the name "prds" refers to the PRDS of that processor. This is most useful when the PRDS is used as a stack during process switching. It is also useful for accessing per-processor data in a uniform manner.

Idle processes take no page faults and never wait. They always run in ring zero. They have no KSTs and have no interaction with the storage hierarchy, other than the handling of interrupts. Idle processes can handle any interrupt that does not involve page faults.

Starting Processors

Having discussed idle processes and their uses, we now proceed to discuss the starting of processors. Traffic control initialization (as performed by the program tc_init) causes the starting of the bootload processor. Dynamic reconfiguration starts other processors. Starting a processor consists of creating its idle process and sending the processor to be started an "initialize" software interrupt to force it to enter that process. What is meant by "starting" the bootload CPU will be explained below.

Creating an idle process consists of allocating an APT entry and setting it to describe a loaded, eligible, and ready process, which is also marked as idle and has the processor required bit set, indicating that only the processor for which it is intended may run it. A process ID is fabricated for it, and it is threaded into the ready list as an idle process. A descriptor segment and PDS are created for it (by plm$hc and plm$makeseg, respectively). The SDWs for each are set in the new descriptor segment and the PDS is initialized from template_pds. The first page of each is wired. The relative AST entry pointers of these two segments are put in the appropriate places in the AST entry of the segment (to find them when it is time to delete the idle process, not for loading, as a normal process). Some per-process variables in the new PDS are initialized. Next, if this is not the idle process of the bootload CPU, a PRDS is created, initialized, and wired and its SDW stored in the descriptor segment of the idle process. If this is the idle process of the bootload CPU being created, the SDW of of the segment prds of initialization is used. In either case, the relative AST entry offset of the PRDS is stored in the APT entry of the idle process.

Once the idle process is thus ready for use, a program called init_processor is called (at the entry point init_processor). This impure procedure stores the DBR of the idle process it is to start in a location in its own text segment. Pointer registers and another pointer are stored too. Next, the absolute addresses of an SCU data area and a sequence of code (init_processor$first_steps) are computed, and absolute-mode SCU and TRA instructions addressed to these locations stored in the interrupt vector for the software initialize interrupt. The current system controller mask for the processor now executing is saved. A mask allowing initialize interrupts is now set as the mask for this processor. An initialize interrupt is sent, via a system controller, to the processor being started (which may be this processor). A loop is entered to await the acknowledgement of this interrupt by the processor started. An error is returned (after restoring the mask and interrupt vector) if no acknowledgement is received after a fixed time. A code indicating successful starting is returned if the acknowledgement was returned in time.

What does a processor, particularly the bootload processor do when given the initialize interrupt? It stores its SCU data at location scu_data in init processor. Still in absolute mode, it transfers to init_processor$first_steps. The DBR is loaded via an IC-format (instruction-counter relative addressing) instruction, using the value saved by the call side of

init_processor. An ITS pointer set up by the call side is indirected through, by a transfer instruction, causing entry into appending mode in the desired idle process. The pointer registers are also loaded from values again stored in the text section of init_processor by the call side. A check is made to see if the processor that caused the processor initialization is the same processor that took the initialize interrupt. This should happen only once for the bootload CPU. A check is made that the processor tag as read from the switches corresponds to the tag put in the new PRDS, derived from the CONFIG deck. If all checks pass, the processor is ready for use. Its mode register is set (for stopping history registers on faults) and its cache is enabled. The value of its timer register is saved in its PRDS (for traffic control calculations). The APT entry of this idle process is set to indicate that the idle process is running. Finally, the flag to acknowledge the successful initialization to the processor that sent the initialize interrupt is set. The bit array (scs$processor) in the SCS, showing which processors are running, is set to indicate that this processor is running. A stack frame is set up for init_processor in the page fault trace array in the PDS of the idle process. Since the idle process takes no page or segment faults, this area is never used. The stack frame must be set up here as this area is wired. This stack frame is necessary for the call/return sequence with the traffic controller at preempt time. Incidentally, all of the above code is inhibited.

The last step taken by a newly-initialized processor before entering its idle loop (check reconfiguration switches, DIS, loop), is to send itself a preempt interrupt. This causes the processor to pass through the getwork routine of the traffic controller to see if there is any more worthy process to run. If there is none, the traffic controller returns to the idle process. (By definition, the idle process is the least worthy process. The traffic controller always chooses the idle process when there is no more worthy process to run.)

Now all of this is quite necessary for the initialization of any processor. For each processor, an idle process must be created and started. As was mentioned before, this is done at initialization time by tc_init for the bootload processor. Dynamic reconfiguration starts all others. Most processors started in this manner were executing a DIS at the time of the initialize interrupt. Their previous state was not meaningful. For the bootload processor at initialization time, however, this is not the case. Its previous state was involved with running system initialization. Somehow, this work must be picked up by someone. This was the entire point of setting up the APT entry of Initializer.SysDaemon.z to indicate a ready process, eligible

and loaded, which has never run. When the bootload idle process preempts itself before entering its idle loop, it finds a more worthy process to run, namely, Initializer.SysDaemon.z. It is ready, eligible, and loaded. Hence, the decision is made to run that process. The flag that says this process has never run causes the traffic controller to pick up pds$last_sp in that process and return into that frame on the PDS. To make this work, the call side of init_processor ingeniously stored its current stack pointer in the back pointer of its own stack frame (after saving it), and in pds$last_sp, and set the procedure return point in its frame to the label where acknowledgement from the started processor is awaited. Hence, when the process of Initializer.SysDaemon.z is picked up, the DBR of initialization is loaded the process of the initializer is set to running, and a return is made into the stack frame set up by the call side of init_processor. Initialization has now become Initializer.SysDaemon.z. Initialization proceeds in ring zero of the initializer process.

The Completion of Traffic Control Initialization

When the bootload CPU has been started, the flag in tc_data (tc_data$wait_enable) that allows the full, normal wait/notify mechanism to function is set. (It is necessary that there be idle processes for this mechanism to function.) Traffic control is now initialized.

USER I/O INITIALIZATION

After tc_init has created idle processes, started the bootload processors, and made the notion of process function, user-accessible I/O is initialized. This consists of the communications control software and the I/O interfacer. It is done after traffic control initialization for no good reason, but just done.

Communications Initialization

The DATANET 6600 FNP control program is called to initialize its data areas at this time. There is no reason why this need be done now. The fact that the DATANET 6600 FNP control program was loaded as part of collection 1 (due to its being wired) seems sufficient reason to initialize it in collection 1. DATANET information is extracted from the D355 configuration cards. Per-DATANET information is stored in the dn355_data segment. This includes mailbox pointers, port and interrupt cell numbers,

and DATANET numbers. The IOM manager is called to assign device indices to each DATANET. They are connected through IOM special channels. This allows data transfer to the DATANET via the IOM. The interrupt handler is not assigned at this time, as the DATANET interrupts not through the IOM interrupt mechanism, but via its own setting of an interrupt cell. LSLA/HSLA indices are also ascertained from the CONFIG deck at this time and stored in the DATANET data area.

There is one program, tty_init, that has the major responsibility for initializing all data bases necessary for the subsequent use of the ring 0 typewriter control software. The primary data base is tty_buf. This is a segment that consists of various control and metering variables in its header. This is followed by one eight word control structure for each communications line specified in the CONFIG deck. The remainder of the segment (whose length in 1024 word blocks may be specified on a config card) is allocated into chained sixteen word buffers. These buffers are shared among all terminals dialed to the system. tty_init is responsible for the initialization of a portion of another data base, dn355_data.

tty_init begins its work by initializing several variables in the header of tty_buf such as the time, the absolute address of tty_buf, etc. Then it reads all of the LSLA and HSLA config cards. These cards are checked for consistency and any errors are reported via syserr. If there are no errors, dn355$assign is called once for each configured line. This will cause a table in dn355_data that maps device index (devx) into physical typewriter line number to be filled for later use in the ring 0 typewriter software. In addition to initializing this table, one eight word entry is made in tty_buf for each configured line. These entries are in an array that is indexed by devx, and follow the header information in tty_buf. In each entry is placed an initial terminal type that is set based on the baud rate of the line. This is so the typewriter software can make some initial decisions when receiving a dialup on a given line. An entry is also made in the IOAM data base by calling ioam_$define_name for each configured line.

Once all of the configured lines have been processed, the last task is to take the remaining unused space in tty_buf and chain this into a threaded list of 16 word buffers. Each buffer has the relative address of the next buffer in its first 18 bits, and a special pattern of alternating ones and zeroes in its last word to mark the buffer as free. The address of the first buffer in the chain is stored in the variable, free, in the header of tty_buf.

THE END OF INITIALIZATION

After traffic control and user I/O initialization, init_collections$init_collection_2 returns to initializer. A call is made to delete_segs$init to delete all initialization segments, including init_collections but not including initializer itself (this is a supervisor segment). Initialization, or more precisely the initialization environment, is now the process of Initializer.SysDaemon.z in ring 0. A call is made to init_proc$multics. init_proc$init_proc is usually the first procedure invoked in a process. It initializes the process KST and makes a pointer, via the linker at the validation level of the startup ring of the process, to the first user-ring program, usually user_init_admin_. It then proceeds to call this program, via an outward call. init_proc$multics, however, calls initialize_kst with a special parameter since the KST for the initializer process was almost entirely set up by the call to initialize_kst with another special parameter by init_root_dir. Hence, the system search rules are set up (the default search directories are created if they do not exist) and the search rules of the initializer (like those of any process) set up following that. A pointer is made to system_control instead of init_admin_. This is the first outer_ring procedure for the initializer. An outward call is made via this pointer to this procedure. Initialization is complete and Multics is operative.

RETROSPECT ON COLLECTION 2

The basic goals of collection 2 initialization are to set up the ability to initiate segments and take segment faults, to access the storage hierarchy via storage system primitives, to set up traffic control and the notion of process, including idle processes, to load collection 3, and to start the Initializer process.

This happens in the following order.

The segment fault mechanism (the trailer segment) is set up. The root directory is accessed and initialized. Supervisor segments that are to go in the hierarchy are put in the hierarchy. All fault handlers are set up. Collection 3 is loaded directly into the hierarchy. The bootload idle process is set up. Initialization is picked up by the bootload CPU in the process of the initializer. The process of Initializer.SysDaemon.z is initialized and started on its way.

SECTION IV

SHUTDOWN

Shutdown consists of the orderly cessation of service of a Multics system. A crash or failure consists of a system problem causing a return to BOS, either via a call to syserr, a system trouble interrupt sent upon detection of a problem, or an operator-initiated execute fault or manual transfer, initiated from the maintenance panel. After a crash, an orderly shutdown can be attempted via an operation known as an Emergency Shutdown (ESD). This operation restarts Multics at a special point and attempts to complete a normal shutdown.

What constitutes an orderly cessation of service? It is the responsibility of the answering service to shed the user load of the system at the time of an operator-initiated shutdown. At the time of a crash, that user load is shed in a disorderly fashion. The responsibility of system shutdown is to ensure the consistency of the storage system. Main memory and the paging device are a buffer for the storage system, which resides totally on disk between bootloads. The goals of an orderly shutdown are to:

1. Drive all pages out of main memory and out of the paging device onto disk.

2. Ensure the integrity of directories--that all branches, specifically file maps, are updated from information in the AST. No branch must indicate that its segment is active or the next bootload will believe it.

3. Ensure the integrity of the FSDCT--that all changes made to the FSDCT (including the file map of the root directory and all changes made during shutdown) are reflected to the disk copy.

4. Relinquish disk storage used by segments not in the hierarchy, i.e., the paged supervisor segments not in the hierarchy.

The responsibility of emergency shutdown is to try to accomplish as much of the above as is possible, in an environment deficient in an unknown way.

Shutdown is somewhat akin to initialization in the sense that the environment in which it runs is gradually depleted as opposed to the continually growing environment of initialization.

Thus, shutdown consists primarily of:

1. Flushing main memory--several times.

2. Deactivating any segments that can be deactivated.

3. Updating the branches of any segments that cannot be deactivated (i.e., entry-hold segments, like those supervisor segments in the hierarchy).

4. Deleting the hardcore nonhierarchy segments via the AST traversal mechanism of initialization.

5. Flushing the paging device.

6. Updating the FSDCT, perhaps several times.

The flowchart, Figure 4-1, shows the sequence of these operations for both normal and emergency shutdown. The module (or entry point) responsible for performing each function is given at the bottom of each box.

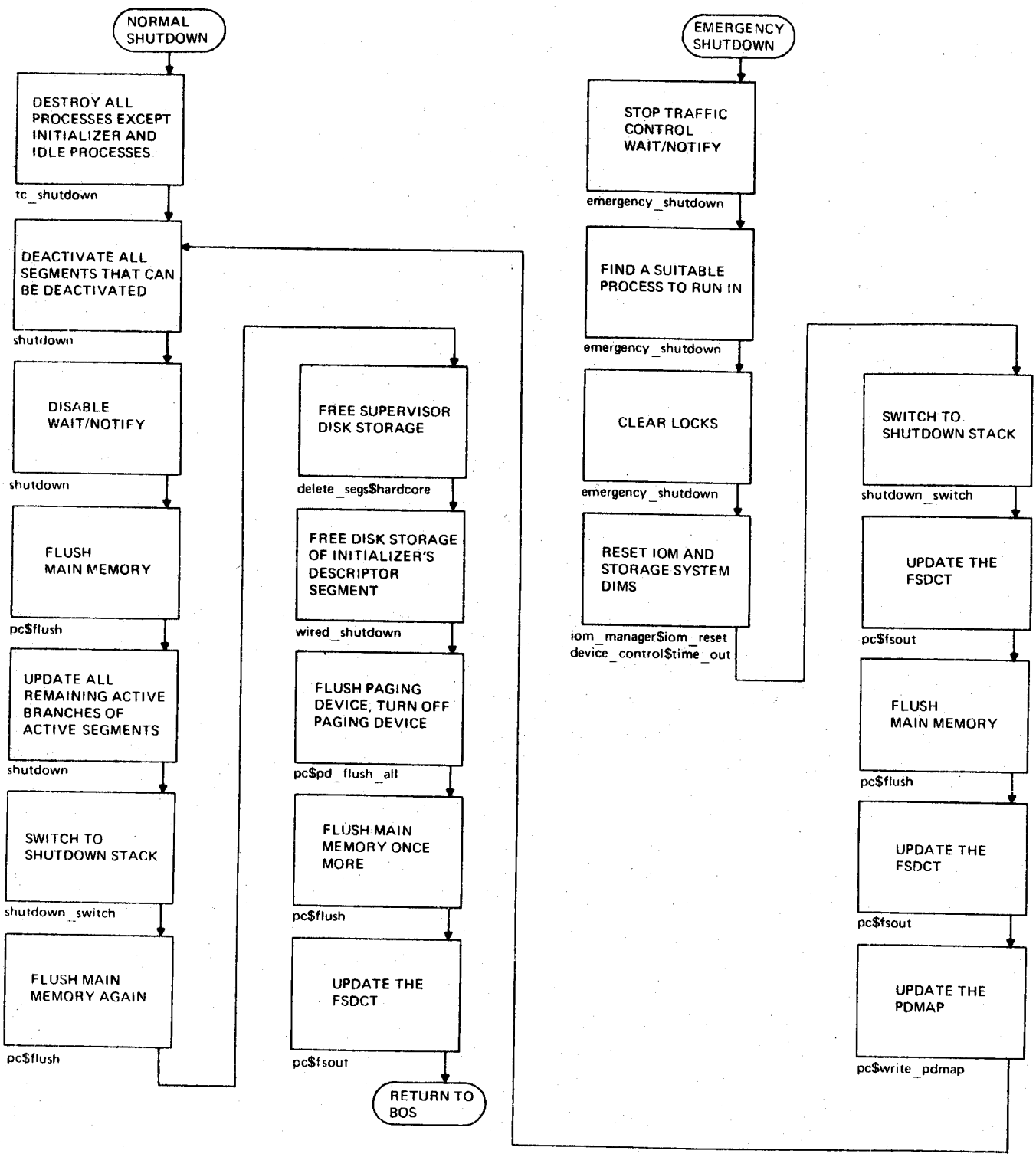

Figure 4-1. Shutdown

NORMAL SHUTDOWN

Normal shutdown is performed by the procedure shutdown and the programs it calls. When shutdown has finished its work, it switches stacks to the segment shutdown_stack and invokes the procedure wired_shutdown. Once in this procedure, no nonwired paged hardcore segments are utilized. This procedure ultimately returns to BOS via a call to pmut$bos.

Shutdown begins with a call to tc_shutdown. tc_shutdown sets the flag tc_data$system_shutdown to 1. This flag changes the Multics locking strategy to allow any process to lock any lock, regardless of whether or not it is already locked. This strategy is based upon the fact that only one process is running. (This applies to only wait/notify type locks, such as directory locks and the AST lock.) This flag also prohibits the depositing of disk records (to be discussed) and modifies the behavior of the utility program wire_stack and the teletype control package.

tc_shutdown continues by calling deact_proc$destroy to destroy all existing processes (save the initializer and idle processes). This destruction is done via the normal process destruction mechanism, which is not graceful. Graceful removal of processes is the responsibility of the operator, before shutdown is invoked. Only the initializer process is allowed to perform a normal shutdown--this check is made at the very onset of the program shutdown.

The next step of shutdown is to deactivate all segments that can be deactivated. Not only does this force their pages out of main memory, ensuring their consistency, but ensures that the branches, including the file maps for these segments, are consistent. The issue of pages on the paging device is left aside for a moment. The existence of such pages does not affect the information in the branches. This deactivation is done as a loop over all of the regular (four sizes of page tables for hierarchy segments) AST lists. The hardcore AST list will be dealt with later. Supervisor segments that are in the hierarchy are on the regular lists, not the hardcore list.

For each AST list, the list is traversed to find each active segment with no inferiors active. If it is not entry-hold active (supervisor segments will be entry_hold active, as will KSTs and PDSs of other processes during an emergency shutdown, and the KST and PDS of the initializer in all cases), it is deactivated. After each segment is deactivated, an inner loop is made, checking its parent, and its parent, and so on, to deactivate them if they now have no inferiors active as the result of the preceding deactivation. As this inner loop proceeds, a check is

made that the next ASTE to be inspected by the outer loop (trying to find any ASTE with no active inferiors) was not the segment being deactivated by the inner loop. The root is special-cased, and not deactivated. Figure 4-2 shows this loop.

When the above loop is finished, the only segments left active on the regular AST lists are those whose entry-hold-active switch is on and their containing directories. The branches for these segments are updated next. First, however, the switch tc_data$wait_enable is set to 0, its value prior to the initialization of traffic control. This disables the wait/notify mechanism and reverts this mechanism to the more primitive mechanism of initialization (see "Traffic Control and Rings" in Section I). Also, pc$flush is called to write out the contents of main memory. This is done at this time to cause all pages in main memory to have device addresses assigned at the time that their branches are updated in the next sequence of calls. (This check is somewhat redundant.)

The AST lists for hierarchy segments are now traversed once more. The routine updateb, which updates branches from AST entries, is called to update the branch of each active segment. This assigns device addresses (redundantly) to all pages still in main memory if they have none, and updates file maps and time used/modified information in the branch. Quota accounting is also updated at this time. Also at this time, the astep fields in the branches of these segments are zeroed. This critical step is the inverse of that performed by init_branches and ensures that the first segment fault on any such segment during the next bootload finds that the segment is <u>not</u> active and must activate it. The salvager also performs this critical operation should shutdown (or emergency shutdown) fail. If neither of these measures succeeds, system failure is almost certain on the next bootload.

Pages in main memory that are zero are not assigned device addresses by page$pwrite, the primitive called by pc$flush. Thus, pages of entry-hold-active segments in main memory that are zero are assigned device addresses by updateb at the time their branches are updated and later paging activity causes those pages to be discovered to be zero and freed by normal paging activity. Were such pages discovered to be zero by pc$flush, the following problem would result. As this depositing was not noted in the already-updated file map, an unprotected address (causing fatal system failure) would be noted at the next bootload when such pages were deleted via segment deletion. The prevention of this effect is the point of disabling page-depositing during shutdown.

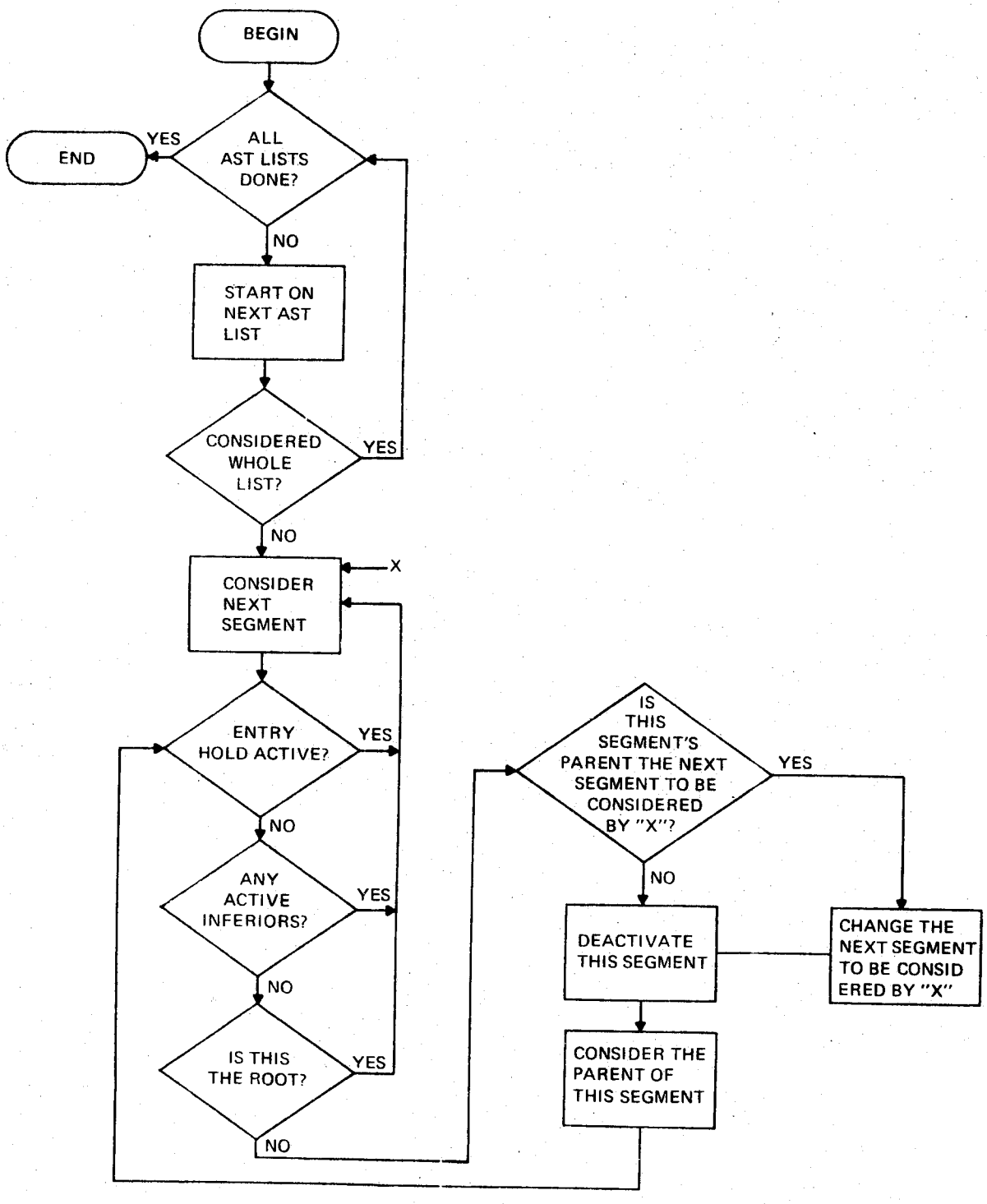

Figure 4-2. Deactivation Loop of Shutdown

When all these steps of shutdown are complete, the procedure wired_shutdown is called via the interface shutdown_switch. The latter interface abandons the current stack (which for a normal shutdown is the PDS of the initializer process and for an emergency shutdown is already shutdown_stack) and establishes a stack frame at the base of the supervisor segment shutdown_stack. The segment shutdown_stack is a segment consisting of zeroes; it is loaded from the MST, but its AST entry is threaded out of the AST lists by make_segs_paged (see "The Making Paged of Segments" in Section II). Its PTWs are marked as "wired" but not in core. shutdown_switch touches all of its pages, bringing them into main memory permanently. A stack header is copied from the current stack and a stack frame set up at the base of shutdown_stack. All code from this point on takes no page faults--all paged, nonhierarchy supervisor segments are deleted. Finally, shutdown-switch invokes the procedure that it was called to invoke, in this case wired_shutdown.

wired_shutdown begins by calling pc$flush to write out all of main memory once more. This is to ensure that changes made by the previous loop in shutdown are reflected to secondary storage. Next, disk LRU metering is turned off as the segment disk_traffic_data is deleted by the next call, which deposits disk addresses (these would cause references to disk_traffic_data if LRU metering were on). A call is made to delete_segs$hardcore. This procedure, delete_segs, is used during initialization to traverse the AST init seg and temp seg lists to delete these types of segments. Here, it traverses the hardcore seg list, deleting all supervisor segments that are not in the hierarchy. Thus, any segment that is paged must be in the hierarchy and wired if it is to be used beyond this point. The point of deleting these segments is to relinquish their disk and paging device storage. If this were not done, this (disk) storage would be unusable until the next "long" run of the salvager. Next, the disk and paging device storage occupied by the descriptor segment of the initializer is relinquished. This does not, however, delete this segment, which is still in use.

Next, the paging device is flushed via a call to pc$pd_flush_all. This causes read-write sequences (rws) for all modified (with respect to disk) pages on the paging device and the freeing of all others. The paging device is then disabled, and another call to pc$flush made to write out all pages that were in main memory, but had copies on the paging device at the time of the call to pcd$pd_flush_all. Such pages were driven off of the paging device by pc$pd_flush_all as the latter noticed that they had copies in main memory, which would be written out. Next, the updated paging device map is written out to the paging device, pc$write_pdmap for the benefit of the salavager.

Finally, the FSDCT is written out, reflecting all changes to device allocations and the root file map during the entire run and shutdown. BOS is then invoked via pmut$bos and shutdown is complete.

pc$flush is called several times during shutdown--before the deactivate loop, after the updateb loop, and after the flush of the paging device. While only the last of these is strictly necessary, the repeating of this step serves as a hedge against failure of any of the intervening steps. This is also done as one of the first steps of emergency shutdown.

Also at several times during shutdown, a variable in the FSDCT indicating the relative success of shutdown is updated. This allows the salvager and BOS to make decisions based upon the relative success (last point passed successfully) of shutdown.

EMERGENCY SHUTDOWN

After an unexpected return to BOS due to a system failure, the operations performed by shutdown must still be performed if the consistency of the storage system is to be maintained. However, the state of the Multics environment at this time is unclear--it is not known which mechanisms are functional and how much so. Thus, many redundant measures have to be taken to ensure the success of as many steps as possible.

After Multics has returned to BOS due to an unexpected error, a DUMP or FDUMP can be taken by the operator. After this, the BOS command ESD can be given to initiate an emergency shutdown. This command alters the machine state saved by BOS at the time of entry to BOS. The segment emergency_shutdown is located by BOS from the SLT. The machine conditions are altered such that a GO (CONTIN) command resumes control at the first word of this segment. Such a command is then issued. The procedure emergency_shutdown assumes control. This procedure is so written that it can be entered in absolute mode if its base address is known. Thus, it first establishes its linkage pointer and a text base pointer from text-imbedded points set up by initialize_faults$fault_init_one. It then enters appending mode.

emergency_shutdown sets the flag tc_data$system_shutdown disabling locking and zeroes tc_data$wait_enable, reverting to the initialization wait/notify mechanism. These measures reduce the dependency on locking mechanisms operating properly. It is not even known that locks were in a consistent state at the time of the crash. The SCS is updated to show that only the bootload processor is running, thus disabling connects sent by page

control. The sys_level (no interrupts other than sys_trouble allowed) mask is set. The APT is then scanned for a process that is loaded and eligible. The process that crashed may have been an idle process, which has no usable PDS, or may be defective in some other way. The crashing process is used only if no other is found. If a usable process is found, an LDBR is done switching into that process. The PRDS SDW is carried along as when LDBR is performed by traffic controller.

Next a stack frame is set up on the PRDS to allow calls to be made to entries in wired code which expect a wired stack. Next, the reconfiguration, AST, page table, and traffic controller locks are forcibly unlocked. They may have been locked at the time of the crash and will never be unlocked otherwise. Directory locks are special-cased by means of the switch tc_data$system_shutdown. Also at this time, the process ID of the running process is changed to 777777777776, so that no lock may ever appear to this process to be locked by it (mylock error). The in_bos flag set by the interrupt interceptor (see the Process and Processor Control PLM) is also reset.

Calls are now made to forcibly reset the operator's console, the syserr logging mechanism, and the IOM manager. Special entries are provided within these mechanisms to forcibly reset possibly inconsistent states at this time. A call is made to device_control$timeout to post any disk status that may be unprocessed. An entry to wired_shutdown, wired_shutdown$wired_emergency, is now called via shutdown_switch. The latter enables and initializes the shutdown stack, switches to it, and calls wired_shutdown$wired_emergency.

wired_shutdown$wired_emergency, running on the shutdown stack, writes out the FSDCT, flushes main memory, writes out the FSDCT once more, writes out the paging device map, and then calls shutdown$emergency, which proceeds with normal shutdown just beyond the point where tc_shutdown is called (i.e., starting with the deactivation loop). The idea of all of these measures is to do each one as early as possible in case the next one fails due to unknown or unpredictable causes. Writing out the FSDCT is very important and very easy. Thus, it is wise to do this before flushing main memory, which is less likely to succeed and less important (An inconsistent FSDCT can cause reused address failures, while inconsistent segment contents is a less fatal problem. Neither is truly acceptable, though). As the FSDCT will likely be modified by writing out main memory, it must be written out before the deactivate/update loop, which is even less likely to succeed. This philosophy prevails during shutdown.

SECTION V

MODULE DESCRIPTIONS

Most of the modules used during initialization and shutdown are intended to be called only once. They perform specific functions that can only be done before certain functions are performed and after certain others. Most of these procedures are invoked with no arguments. It is impossible to describe these procedures in module descriptions. Any comprehension of their purpose or function must be gained by understanding them in context. Hence, the names of these modules are given below, with a brief description of what they do and references to earlier sections for a full understanding of their function.

Some modules, specifically init_processor, make_branches, start_cpu, prds_init, shutdown_switch and the prelinker, can be called more than once, but their function is again highly specialized, and not of general utility. Descriptions are included below.

SPECIALIZED MODULES

bootstrap1.alm	accepts environment from BOS. Sets up segmentation, loads collection 1 into unpaged segments.
bootstrap2.alm	sets up stacks, calls prelinker. Creates PL/I environment.
build_template_pds.alm	makes a template PDS for process creation. PDS contains stack frame for return to init_proc.
bulk_store_init.pl1	initializes bulk store mailbox.

clock_init.pl1	ascertains time zone and delta from GMT from CONFIG deck.
collect_free_core.pl1	adds unused pages of main memory to paging pool.
disk_init.pl1	initializes disk control routines, establishing their communication with IOM manager.
dn355_init.pl1	sets up DATANET 6600 FNP variables and IOM manager communication.
emergency_shutdown.alm	accepts control from BOS for emergency shutdown. Creates environment where much freedom is allowed.
free_unused_pages.pl1	adds unused portions of unpaged segments to paging pool.
init_branches.pl1	places those supervisor segments to go in the hierarchy in the hierarchy. Initializes >pdd, other sons of root.
init_collections.pl1	dispatches initialization calls.
init_hardcore_gates.pl1	stores text-imbedded link pointers in hardcore gates, sets up special SDWs for fault restart programs.
init_processor.alm	starts a CPU. Contains first code executed by a CPU and code for idle process.
init_root_dir.pl1	makes root directory into a legitimate directory.
init_sst.pl1	sets up core map, PD map, AST.
init_str_seg.pl1	makes free list of system trailers in str_seg.
init_sys_var.pl1	sets up random system variables.
initialize_dims.pl1	creates or accesses FSDCT, dispatches device initialization calls. Sets up paging.

initialize_faults.pl1	sets up fault and interrupt vector ITS pointers and text-imbedded pointers.
initialize_kst.pl1	used by process initialization. Used by system initialization to allow segments to be initiated by setting up the KST of the initializer. Also sets up search rules.
initializer.pl1	permanent supervisor segment that dispatches initialization calls, mainly to init_collections and delete_segs.
iom_data_init.pl1	sets up IOM mailboxes and control words. Sets up overhead channel handling.
load_system.pl1	reads collection 3 into the hierarchy.
make_branches.pl1	recursively creates the storage system branch for segments loaded from the MST.
make_segs_paged	makes paged segments of unpaged ones. Sets up root and other special ASTEs. Formerly called update_sst_pl1.
prds_init.pl1	sets up PRDS for a processor.
pre_link_1.alm	prelinker driver. Scans linkage sections for links to be snapped.
pre_link_2.alm	snaps a given link.
scas_init.pl1	initializes configuration data about system controllers. Sets up system controller addressing segment (SCAS).

delete_segs

Name: delete_segs

This procedure traverses AST lists, deleting all segments on that list. This deletion consists of calling pc$truncate to hand back the disk and paging device storage occupied by the segments. SDWs for these segments are zeroed as well. There are three entries. No arguments are needed by these calls.

Usage

```
declare delete_segs$temp entry;

call delete_segs$temp;
```

causes all temp segs to be deleted. Used at the end of the initialization of each of collections 1 and 2.

```
declare delete_segs$init entry;

call delete_segs$init;
```

causes all init and temp segs segments to be deleted. Used at the end of initialization. This call is made by initializer, a supervisor segment.

```
declare delete_segs$hardcore entry;

call delete_segs$hardcore;
```

causes all supervisor segments not in the hierarchy to be deleted. Used by wired_shutdown at shutdown time.

find

Name: find

This utility module is used to locate selected cards in the CONFIG deck.

Usage

```
declare find entry (char(4) unaligned, ptr);

call find (name, p);
```

where:

1. name is the name of the type of CONFIG card sought. (Input)

2. p is both input and output. If given as null, the configuration deck is searched from its beginning. Otherwise, it is searched from the point pointed to by p. As a return value, p points to the first card image of the type required, having searched from the required point. If returned as null, there are no more cards of that type.

Examples

```
                i=0;
                p=null;
more:           call find ("cpu", p);
                if p=null then go to no_more;

                i=i+1;
                go to more;
no_more:        /* i contains the number of "cpu" cards */
```

freecore

Name: freecore

This procedure is used to explicitly add a page frame of main memory to the paging pool. It should be used only for such page frames as were not in it at the time of the call. It is used during reconfiguration and initialization.

Usage

```
declare freecore entry (fixed bin(17));

call freecore (n);
```

where n is the number of the page frame to be freed, i.e., 3 means the block at address 6000 octal. (Input)

```
declare freecore$reserve entry (fixed bin(17));
```

This entry is like freecore. However, if the page frame being freed is in abs_usable memory, it is not marked as abs-usable. This prevents I/O buffers from using it.

make_sdw

Name: make_sdw

This procedure is used to create an AST entry (including page table) for a segment on the MST, thread it into an appropriate AST list, and return an SDW describing that segment. It is used by make_segs_paged for collection 1 paged segments and segment_loader for all collection 2 segments.

Usage

```
declare make_sdw entry (fixed bin(18), fixed bin(71),
     ptr, ptr);

call make_sdw (segno, tsdw, astep, ptp)
```

where:

1. segno — is the segment number of the segment for which an ASTE is constructed. This segment number is used to access the SLT. (Input)

2. tsdw — is an SDW using the newly-created page table. This can be placed in the descriptor segment using appropriate calls.

3. astep — is a pointer to the ASTE created. (Output)

4. ptp — is a pointer to the page table created. (Output)

make_sdw determines the proper size AST entry from the max_length and cur_length fields of the SLTE. The TBLS card overrides both of these. The appropriate list on which to thread the AST entry is critical. It is determined by the following algorithm:

if unthreaded entry, or the segment has wired pages, then not threaded at all.

ELSE if slte.temp_seg is on, then threaded on the temp_seg list.

ELSE if slte.branch_required is on, then threaded on the regular AST list with AST entries of this size.

ELSE if slte.init_seg is on, then threaded on the irit_seg list.

ELSE threaded on the hardcore list, aste.hc turned on.

aste.ehs and aste.hc_sdw are turned on in all ASTEs, except in the unthreaded case.

make_sdw$unthreaded is called in the same way as make_sdw, but causes the creation of an unthreaded entry.

tape_io

Name: tape_io

This procedure is called to perform physical tape I/O on the MST. It is called by tape_reader.

Usage

```
declare (tape_io$init_tape, tape_io$final_tape) entry;

declare (tape_io$get_unit, tape_io$get_unit) entry
     (fixed bin(6));

declare (tape_io$read. tape_io$backspace, tape_io$rewind,
     tape_io$unload, tape_io$skip_file, tape_io$set_density_800,
     tape_io$set_density_1600) entry fixed bin (5);

declare tape_io$tape_interrupt entry (fixed bin(12), fixed bin(12),
     fixed bin(71), fixed bin(3));
```

call tape_io$init_tape	sets up the MST reading package. The PCW left by bootstrap1 in physical_record_buffer is inspected.
call tape_io$final_tape	closes the package. The tape channel is marked and deassigned.
call tape_io$get_unit (unit_no)	extracts the current tape unit number from the PCWs being used by tape_io.
call tape_io$set_unit (unit_no)	sets the current tape unit number to be used by tape_io.
call tape_io$read (status)	starts a read into physical_record_buffer. Major status is returned through status.

APPENDIX A

abs-segs

The concept of an abs-seg is used many times during initialization and plays a critical role in the procedure make_segs_paged. For those who are not familiar with this concept, we provide here an explanation of the use and construction of abs-segs.

A program running in the Multics hardcore, including initialization and shutdown, has access to the descriptor segment it is using. It is therefore possible for a hardcore program to construct an SDW pointing to any legitimate page table or contiguous region of main memory. This SDW can be stored at any place in the descriptor segment, and the segment thus pointed to can be referenced via the segment number describing that descriptor segment slot.

Furthermore, the meaning of pointers and symbolic references to that segment number change as the SDW is changed. The segment described by that segment number takes on different identities as the SDW is changed. It is not any given segment at all, but different ones at different times. The segment of changing identity assigned to that segment number is known as an abs-seg. The reserving of segment numbers for such use is valuable as it allows symbolic references to be made to the abs-seg, which in fact reference different segments as the SDW is changed.

Two examples of abs-segs follow.

Page control must check for zero pages of main memory, when it is time to write a page out. It is not known if the page belongs to a segment that is known in this process or not. Hence, page control constructs an SDW describing that page only and places it in the descriptor segment position for the segment abs_seg1. Now, page control need only check the first 1024 words of abs_seg1 to see if they are zero.

Segment control searches the AST for an AST entry to preempt when one is needed. It decides to deactivate a given segment but must update the branch in the containing directory of the segment. An AST entry contains a relative pointer to the AST entry of the containing directory of its segment. Thus, segment control fabricates an SDW describing the page table in the AST entry of the containing directory and places it in the descriptor segment position for the segment dir_seg. A pointer to dir_seg is now passed to the branch updating routine, as a pointer to the containing directory of the segment. Neither the segment being deactivated nor its containing directory need be known in the current process.