Project MAC
Computer Systems Research Division

February 14, 1975
Request for Comments no. 68

ON MULTICS SYSTEM INITIALIZATION

by Allen W. Luniewski

In this paper the initialization of Multics is discussed. The current method is presented briefly and its shortcomings are pointed out. A new method of initializing Multics is proposed. It seems to be a cleaner, easier to understand and maintain method.

In this paper I shall discuss the initialization of Multics. My concern is with that part of initialization which occurs between the transfer of control from BOS to Multics (currently in the form of "bootstrap1") and the time Multics first leaves ring 0. I am not concerned with any possible initializations performed after Multics leaves ring 0 as by then we are in a "real" Multics environment. This paper will cover three areas. The first part will be a discussion of how initialization is done today and some observations on what I perceive to be wrong with it. In the second part a discussiof will be made of general ways to simplify initialization so as to correct the problems perceived with it. The remainder of the paper will deal with a proposed scheme for initialization.

Motivation

The initialization of Multics is an involved operation and currently it does work. As the inner workings of Multics change so must the system initialization routines. When these changes are made, care must be taken to obey the implicit conventions about what can and can not be done at a given time during system initialization in order to keep initialization working. The net effect is that modifications to initialization routines are frequently hard and the routines themselves hard to understand. Another motivation is one of certification. The current method of initialization, because of its complexity, seems very hard to certify. A more certifiable method seems desirable.

Initialization Today

A brief summary of how initialization is performed today will now be presented. A more detailed summary is presented in Appendix A and an involved discussion is available in the Honeywell PLM AN70. The initialization of Multics is basically a bootstrapping process. BOS first reads in a small program (which in fact is the first part of "bootstrap1") from the MST (Multics System Tape or bootload tape) and transfers to it. This program proceeds to set up a very primitive environment (absolute addressing, no external interrupts, no legal faults). It then reads in another program (actually the rest of "bootstrap1"). This program sets up a more complicated environment, reads in additional programs and transfers to them. In this way the entire ring 0 environment is ultimately built. The major milestones passed (in order) during initialization are:

1. Establish an appending environment
2. Set up the stack mechanism
3. Enable symbolic references (establishing the PL/I environment)
4. Initialize the SST (System Segment Table)
5. Bring up the disk storage system
6. Enable the correct handling of page faults
7. Set up a KST (Known Segment Table)

8. Access (construct) the root for a warm (cold) bootload
9. Segment faults and the file system now work
10. Initialize the traffic controller
11. Initialize the teletypes

These eleven items present in a brief way the major path taken by initialization.

Problems With Initialization Today

To complete our look at the way system initialization occurs now, we now will see what makes it hard to understand. Three areas come to mind. First is the sheer volume of system initialization. During the course of initialization almost all of the data bases in ring 0 are initialized. This results in a large number of initialization routines. This type of complexity is inherent and related to the complexity of the supervisor. The large number of routines has a compounding effect on the next two difficulties. A second problem is understanding why things are done in the particular order that they are done in. It takes a careful reading of the documentation and programs of initialization to understand which parts of the standard ring 0 environment are working at any given point during initialization. This makes it difficult to write and maintain initialization routines. The last problem concerns the operations performed at initialization time. One can view initialization operations as operators on the environment - they transform initialization from one environment to another (presumably closer to the ring 0 environment present when Multics is running). The problem with these transformations is that they are from one non-standard environment to another. This makes establishing the correctness of these transformations hard. An alternative way of looking at the last two difficulties is the following. Initialization routines must be very careful not to touch, either directly or indirectly via a fault, interrupt or call, any data base or program which has not yet been initialized or which is in the process of being initialized. These seem to be the major problems with understanding how initialization is performed today.

Solutions

These difficulties with initialization suggest three ways to simplify initialization so as to increase its understandibility. The first, and most obvious, simplification is to have initialization run, as much as possible, in a well defined environment. The logical choice for this environment is the standard ring 0 Multics environment. By running in this environment, the problem of writing initialization procedures is little different from writing ordinary ring 0 code. The amount of time between initialization first being "called" by BOS and when the standard ring 0 environment is working should be minimized, thus maximizing the time spent in the standard ring 0 environment. This can be done by moving current initialization operations to one of two places. First they can be moved to MST generation time. This amounts to making bindings at the time the MST is generated. These bindings can be such items as segments to segment numbers and links to segments as well as bindings relating to a minimal hardware configuration (in terms of Roger Schell's thesis (Project MAC TR-86) these are bindings to logical hardware resources). Second they can be moved to after the standard ring 0 environment is working. This amounts to an assertion that certain operations currently considered necessary to proper Multics operation are not in fact necessary. The final simplification (which is related to the second) occurs by taking a different viewpoint of many initialization operations. Instead of regarding them as one time only acts, we can look at them as specific instances of reconfiguration operations. Of course these operations must run in a standard ring 0 environment, but if we can cause them to, we have achieved a simplification of initialization. I feel that if these changes can be made, initialization will be much easier to understand, maintain and certify. The remainder of this paper will discuss a way to achieve these simplifications.

A New Paradigm for Initialization

The basic course of the proposed initialization scheme is a four phase process. Phase one occurs at MST generation time when a core image of Multics is generated. This core image is generated by making suitable assumptions about the hardware configuration and by making certain bindings. The nature of these assumptions will be discussed later. At bootload time this core image is loaded into core and control passed to an appropriate spot within it. The second phase now occurs. At this time a small number of initializations will be performed to get the core image working as some standard environment. After establishing this environment the third phase proceeds to establish the normal ring 0 environment. As shall be seen later, this principally involves bringing up the file system. The fourth, and final, phase involves invoking reconfiguration routines to adjust to the actual desired hardware and software configuration. For instance, one would dynamically add the paging device instead of demanding it to be there before the standard ring 0 environment is running. On the software side, one might dynamically adjust the number of AST entries of each size instead of fixing the number for all time as part of bringing up the standard ring 0 environment. By proceeding in this way, via a core image and reconfiguration operations, we eventually get to the desired Multics configuration.

<u>Generating the Core Image - Assumptions</u>

The remainder of this paper will concentrate on the core image and how it should work. The assumptions one must make in order to generate the core image represent the first major issue. These assumptions should be few in number in order to increase the flexibility available at bootload time. These assumptions can be of either a hardware or software nature. Hardware assumptions are assumptions about the physical configuration. The number of these assumptions should be small for if too many assumptions are made, the number of installations the MST can be used at is reduced. We propose to make assumptions about the amount of memory available and about the layout of memory. The assumptions about the layout of memory will be of two sorts. First we will assume that we have sufficient low order memory for our needs. This replaces the assumption currently made that sufficient memory is available but that it can appear in two pieces. Our assumption is simpler than currently made although slightly more restrictive. The second assumption about the layout of memory is somewhat more involved. We will assume the absolute core locations of the interrupt and fault vectors, the BOS toehold, the CONFIG deck and the load point of the core image. Currently these are good assumptions - these locations are not changed. If they were to be variable, it would be possible to allow the program which loads the core image to perform these relocations (note that if too much of this sort of "adjustment" is allowed, the resulting paradigm will be similar to the current one and correspondingly hard to understand). A third assumption, also made by the current initialization paradigm, is that there no locations "reserved" above the load point. "Reserved" locations include fault and interrupt vectors, the BOS toehold, the CONFIG deck and the various mailboxes. These seem to be the only hardware assumptions we must make to generate the core image.

Software assumptions should also be minimal. The binding of segments to segment numbers is clearly necessary. By making this assumption, we are able to create segment descriptor words (SDWs) and page table words (PTWs) for the core image as well as for the rest of the ring 0 and initialization routines. We are also able to perform prelinking at MST generation time (bind links to segments). If we were not to do it here, it would have to be done as part of initializing the core image. This is because a linkage fault within ring 0 is not permitted in Multics. If such linkage faults were to be allowed, we still could not allow them early in initialization as the handling of linkage faults requires a hierarchy which is not present initially. No other software or hardware assumptions seem necessary. It should be noted that by making more assumptions the core image can be brought closer to the point where it wakes up as a fully functional ring 0 environment, possibly reducing the time spent initializing the core image and performing initial reconfiguration operations.

The Environment Supported by the Core Image

Another issue is the capabilities of the core image. The goal is to get it to act as much a possible like the standard ring 0 of Multics as soon as possible. We can get close to the ideal of waking up in a standard ring 0 environment. In fact the environment the core image supports is much like that proposed for an H-process by Bob Mabee in MTB-150. We will discuss this by looking at various parts of the ring 0 supervisor.

The first area is interrupts. When the core image wakes up, it knows of no external devices as they will be reconfigured onto the system later. Thus there are no valid external interrupts and the interrupt masks are simply set to reflect this - all external interrupts are masked off. There is another sort of interrupt though - software generated interrupts (such as stop, preempt, IPS (Interprocess Signal), processor initialize, system trouble and syserr log). These interrupts must be enabled at the appropriate system interrupt levels. This can be done either at the time the core image is generated (by binding these interrupts to interrupt cell numbers and interrupt levels) or by determining these assignments as part of initializing the core image. If the first choice is taken, reconfiguration primitives must be provided to change these assignments dynamically. This is necessary so as to allow external devices to use the cells preassigned to the software process interrupts. I choose the first alternative (binding at core image generation time with reconfiguration operations) as it provides more overall flexibility than the second as well as reducing the amount of core image initialization which is necessary.

The second area is that of faults. It appears that all the faults can be handled in their normal manner. Four types of faults deserve special attention - linkage, segment, page and bounds faults. Linkage faults should not occur during normal ring 0 operation so the occurrence of such a fault in the ring 0 core image should cause the system to die. If linkage faults were to become legal within ring 0, we still could not permit them since the handling of a linkage fault requires a hierarchy and file system which won't be present when the core image wakes up (more on this later). Segment faults can be handled in their normal way, however, such faults should not occur as all needed segments will be made active at the time the core image is generated. Note that segment activation and deactivation will not occur unless a segment fault occurs and we are carefully avoiding causing such faults. In this way we will not introduce "new" segment faults after we start the core image running. If a segment fault were to occur, we would be unable to handle it as the KST would be almost null - no segments except the root are known to the process. Bounds faults must be carefully handled. Since we may take bounds faults on segments which have no branches in the hierarchy, the bounds fault handler must take care not to attempt to modify the segment's file map as it does not have a branch. Under the proposed new storage system

(MTB-110) this problem will not occur (file maps will always be full size and are not modified until segment deactivation time). We will discuss page faults in a later paragraph.

The various file system primitives are the third area for consideration in regards the capabilities of the core image. When we first come up in the core image, we have no hierarchy so that all file system operations lack meaning. We can prevent explicit operations by auditing initialization routines and suitable programming conventions. Implicit use is harder to stop. Two implicit operations come to mind - quota checking and returning pages to the file system. Quota checking can be turned off by a switch in the AST entry for the segment (this ability currently exists). Returning of pages (which occurs implicitly when page control detects an all zero page) can be prevented by setting another switch in the segment's AST entry (this ability is also currently present). If some file system operations should occur, the system should die gracefully. This can be done by noting that all file system operations depend on being able to access the root. We set the root's PTWs so that a page fault occurs when the root is referenced. The page fault handler will then call a device driver to read the page in. This driver, an initialization routine, traps on the read attempt and causes the system to crash. Thus until the system is ready for file system operations to occur they will be prevented from occurring by program auditing and programming conventions and if they do occur they will cause a controlled system crash.

The above three areas seem to be the crucial ones in dealing with the core image. All other components of ring 0 should work correctly. In particular, there seems no reason why traffic control can not be working at the instant the core image comes alive, thus allowing multiprogramming. In summary, the core image should have most of the capabilities of a standard Multics ring 0. Most operations work properly. One set, the file system, definitely do not work.

<u>Core Image Initialization</u>

The next topic for discussion is what must be done between the time the core image wakes up and when it provides its restricted environment. It is during this time that we are running in an incomplete environment, so we want the operations which must be performed to be simple and as few as possible.

Nine operations seem necessary to get the core image working. Everything else which is currently done at initialization time will either be done at MST generation time or as a reconfiguration operation after the core image is running. These operations must be done at bootload time as they are inherently configuration dependent. Performing them at MST generation time, while possible, involves making additional configuration assumptions. These additional assumptions serve to severely restrict the possible configurations the system can boot on.

When first transferred to by BOS there are two things the core image must do very quickly. First an appending environment must be established. This involves finding the absolute location of the descriptor segment (or its page table as the case may be) and loading the DBR (Descriptor Base Register) apppropriately. The second important action is to save any information passed to Multics by BOS. Currently BOS passes the absolute core locations of bootstrap1, the IOM mailbox and the interrupt vector as "parameters" to Multics.

Having performed these initial operations, it is necessary to perform some configuration dependent operations. First pointers must be set to point to the clock in the bootload memory, the presence of a clock being essential to correct Multics operation. Second the SCAS (System Controller Addressing Segment) must be constructed. This is a segment with a "page in every memory". Initializing the SCAS is closely tied to the third initialization, memory initialization. All of the memories that the core image lies in must be "configured" in. Port addressing words words for these memories must be set up as well as adding unclaimed pages of these memories to the free core list. These memories must be special cased in this way as the normal reconfiguration primitive to add a memory does not know how to handle a memory that is already partially in use. A fourth necessary operation is the construction of the channel masks. These should be made to reflect our initial, minimal configuration of one CPU (the bootload processor) and the system controllers we are using. No other modules are part of the system configuration so that the channel masks should only allow communication between the bootload processor and the configured system controllers. Note that setting the channel masks in this way prevents communication between the bootload processor and the bootload tape. Establishing this communication link is part of initializing the bootload tape, the fifth initialization. We must initialize the bootload tape as it contains most of the system programs. It is an initialization, as opposed to a reconfiguration, operation since page faults, as discussed later, will be handled from the bootload tape. To enable the communication link between the bootload tape controlling program and the bootload tape (via its IOM) the channel masks must be modified. This could be done in a brute force, ill-structured, way simply by modifying the mask. A better way, however, is to use a reconfiguration operation to add the IOM to the system (causing a structured change to both the channel masks and the interrupt structure - masks and handlers) and then use the IOM manager to add the bootload tape as a device. Doing it in this way is much cleaner than the brute force approach as it uses normal operators to do the necessary work. Of course care must

huh?

be taken to insure that the necessary programs to perform these operations are part of the core image.

Three initializations remain to complete the initializing of the core image. First the six process interrupts must be set up. Masks and simulate patterns for them must be constructed. The options for doing this have previously been discussed. Second the operators console must be initialized, again by setting up the communication path through an IOM (adding the IOM to the system if it is not already present) and then using the IOM manager to add the operators console as a device. The last initialization is a minor one. The time zone the system is running in is ascertained as well as the time difference from GMT. With this initialization, the standard environment provided by the core image is running.

Although it has taken a long time to discuss these initializations, they are not that involved. There are only nine of them, and all of them are relatively simple. Much talk has been devoted to the idea of running in a known environment, so that a short discussion of the initial environment, after setting up the core image as described, is appropriate. It is a paged, segmented environment. The stack environment is in operation (allowing the normal call - return sequence). Symbolic references (through links) are possible. Communication with the outside world (such as IOMs and system controllers) is possible

How much simpler would it be if you created an MST for a specific installation?

through normal means, however only a limited number of external devices are known. Page faults can be taken and will be handled from the bootload tape. Traffic control is in operation. Within this restricted environment the remaining initialization procedures will run.

Page Faults

A number of times during this paper the proposal to use the bootload tape as a read-only source for satisfying page faults has been mentioned. A more detailed discussion of this proposal will now be presented. Page faults will first be handled during the third phase of initialization, bringing up the file system. The routines we will be faulting on will be those directly associated with phase three. The first issue is why bother to handle page faults at all. Three reasons come to mind for handling these faults. The most persuasive argument is that the system normally does handle these faults. To forbid these faults, by refusing to handle them, serves to increase the time spent in a non-standard environment, thus violating one tenet of this proposal. A second reason for allowing these page faults is one of programming ease. If they are not allowed, care must be taken when calling a program to be sure that it is in core. This places needless burdens on the programmer. The last motivation is to minimize core usage during initialization. By being able to read read-only pages from the bootload tape we reduce the core requirements for bringing up Multics, enabling Multics to come up on a smaller configuration. The core tradeoff is significant(*). The core image, if page faults from the tape are allowed, will be from 80K to 100K in length. Not allowing such page faults, thus adding the file system routines to the core image, adds about 65K to the size of the core image. This represents the difference between being able to boot on a 128K system and not.

The use of a tape as a source of pages to satisfy page faults is admittedly a strange idea. Page faults are generally random implying random access to the storage system. A tape is, however, a sequential access storage medium. Its use as a storage system seems to imply a lot of tape motion - forward spacing (or skips) and backspacing. We would like to minimize such motion since it is not an efficient way to use a tape. There are two important facts which will allow us to minimize such motion. They are the fact that the real storage and file systems will be among the first subsystems brought up and two, that initialization takes essentially the same path each time it runs, allowing optimal placement of records on the tape.

After initializing the core image, one of the first things which will be done will be to bring up the storage and file systems. This will allow branch creation, segment activation and

(*) The quoted figures are approximations based on system 24.7. They should be regarded as "ballpark" figures only.

deactivation, and the initiating and terminating of segments. Most importantly it allows segments to reside on disk. Once the storage system is up, and knowing the order of pages on the tape, it is a simple matter to copy, sequentially, the segments on the tape to the real storage system, thus eliminating the use of the tape as a storage system from then on. While bringing up the storage system page faults will be handled from the bootload tape. These could even be eliminated by placing the programs necessary to bring up the storage system in the core image.

The paging from the tape which may be necessary to bring up Multics is not objectionable but the wasted tape motion involved is. We can observe that initialization takes essentially the same path through its programs and data each time it runs (for a given MST that is). We can take advantage of this by tracing the page fault behavior of the system while it faults to the tape. We then can rewrite the tape in an optimal pattern so as to minimize tape motion. This technique, when combined with the observation on bringing up the storage and file systems, and the need to satisfy page faults justifies the use of the bootload tape as an initial, temporary, read-only storage system.

A minor issue raised by using the MST as a read-only "storage system" is how to organize the tape. I propose to organize the MST into three files. The first file will contain the core image and any other data needed by the core image loading program. The second file will contain the non-core resident pages of the initially active segments. In other words it is our read-only storage system, and we will handle page faults from it. The third file will be analogous to the current collection three on the MST. It will contain those segments which go into the hierarchy but are not needed to bring up the system. One might think they could be a part of the second file. This however would require the segment to be active, creating a very large number of active segments and a correspondingly large AST. Since the segments in the third file are not needed to bring up the system, this large AST does not seem justified, thus the three file organization.

The Remainder of Initialization

Having now spent a considerable length of time discussing core image related problems we can quickly deal with the rest of initialization. Once we have the core image up and running we can now proceed to upgrade it to a standard Multics ring 0 environment. The one hole in the environment provided by the initialized core image is the lack of the storage system. Thus the next set of operations to be performed are those needed to bring up the storage system. IOMs and storage system devices must be added to the system using reconfiguration primitives. The root must then be accessed (or constructed if a cold boot), lists of records on the devices accessed (or created) and branches created for those segments (on either the second or third file of the new MST) which need them. Once the storage system is up, the complete ring 0 environment is present. Reconfiguration primitives are now invoked to adapt the software to the desired configuration. Additional units of memory, a paging device, line printers, teletypes and the various other pieces of hardware present in the actual configuration are added to the system. Reconfiguration operations are also invoked on the software. The number of AST entries of each size is changed as desired, the syserr logging mechanism is started, traffic control tables are brought up to size, traffic control parameters reset and various other software reconfigurations are performed. Appendix B has a list of reconfiguration primitives which seem necessary. After performing a suitable number of these reconfigurations the system is up and running.

Remaining Issues

In this paper a number of ideas and proposals have been presented. However three issues have not been discussed. First is the issue of generating the new MST (and most importantly the initial core image). An appropriate analog to the current "header file" must be developed as well as a new "generate_mst" program. A second issue is the loading of the core image. How should it be done? By whom? How do we protect against the fragility (susceptibility to bit errors) of the core image? The third unanswered issue is that of the various reconfiguration operations. How hard are they to build? Are they desirable in a more global context? These issues are important but not central to the issue of technical acceptibility of this paradigm.

Conclusion

This paper has attempted to present three major issues. First, how is initialization done today and why is it hard to understand. Second, in a general way what can be done to make the initialization of Multics easier to understand. Third, a new paradigm has been presented and discussed for initialization. Hopefully the important issues have been raised and discussed clearly.

Appendix A

Initialization Today: A Summary

This appendix contains a time ordered list of the major calls made during system initialization. It contains a list of programs and a brief description of their function. The programs are listed in the order that they are called.

1. bootstrap1

   Read collection one. Establish an appending environment. Set the interrupt vector to ignore all interrupts and faults.

2. bootstrap2

   Set up a stack frame on the segment "pds".

3. slt_manager

   Initialize itself. Return to bootstrap2 the segment numbers of "pre_link_1" and "pre_link_2".

4. pre_link_1

   Pre-link collection one by "snapping" all links that can be snapped at this time.

5. initializer

   A call dispatcher.

6. init_collections$init_collection_1

   A call dispatcher.

7. initialize_faults$fault_init_one

   Set sys_info$clock and prds$proc_contr_ptr to point to the bootload memory. Lockup and timer_runout faults are transferred to "wired_fim$ignore" which ignores the faults. Page and connect faults are directed to their correct handlers. All other faults are ignored by an SCU-RCU instruction pair. Pointers are stored in the text of some programs ("ii" and "fim" are two examples) as these programs can not generate them when they are called.

8. iom_data_init

Initializes the IOM mailbox and the IOM manager. All IOMs are added to the system.

9. oc_data_init

The operators console is added to the system and initialized.

10. dn355_init

Per-Datanet information is stored for each Datanet communication computer in the system. HSLA (High Speed Line Adapter) and LSLA (Low Speed Line Adapter) indices are stored for future reference.

11. scas_init

Verify the configuration as specified in the CONFIG deck. Set up the SCAS (System Controller Addressing Segment).

Appendix A

Initialization Today: A Summary

12. scs_init

    Port addressing words are set up. The system controller - control processor relationships are set up for all CPUs and MEMs in the system. Interrupt initialization is performed (channel masks, interrupt masks, simulate patterns and interrupt handlers are set up). Interrupt cell assignments are made.

13. initialize_faults$interrupt_init

    Most interrupts are directed to their final destinations. Per-processor information is set up in the segment "prds".

14. clock_init

    Determine the local time zone and initialize relevant variables.

15. trace_init

    The system debugging facility (to either magnetic tape or a line printer) is set up.

16. init_sst

    Initialize the segment "sst" by setting up the header, core map, paging device map and hash table, AST and page tables.

17. initialize_dims

    Find the master device. Determine the partitioning of the disks in the system. Add the bulk store and disks to the system. Access (or create if a cold boot) the FSDCT. Set up the segment "pdmap_seg". Set up the paging device used list.

18. priveleged_mode_init$update_sst

Makes paged all segments which are currently unpaged but which should be paged. Create a paged descriptor segment. Allocate an AST entry for the root. Collect all free core not already known about.

PAGE FAULTS NOW WORK.

19. syserr_log_init

Initialize the operators console logging mechanism.

20. delete_segs$temp

Delete all unneeded collection one segments.

21. debug_check$copy_card

Set various system wide debugging options.

22. tape_reader$init

Initialize the collection two tape reader.

23. segment_loader

Load collection two and then use "pre_link_1" to pre-link collection two.

24. init_collections$init_collection_2

A call dispatcher.

25. init_str_seg

    Initialize the AST trailer mechanism.

26. init_hardcore_gates

    Initialize the hardcore gates by storing pointers to their linkage sections in their texts.

27. build_template_dsegs

    An obsolete data base is constructed.

28. get_uid$init

    Initialize the storage system unique ID generator.

29. init_sys_var

    The idenity of the initializer console is determined. The device table is initialized to the null case (no devices). The time of bootload is determined. Some error codes are stored in the segment "sst".

30. init_root_dir

    Sets up a KST (Known Segment Table). The root is made known. The root is constructed if this is a cold bootload.

    SEGMENTS CAN NOW BE CREATED, DESTROYED, INITIATED, TERMINATED, ACTIVATED AND DEACTIVATED. SEGMENT FAULTS NOW WORK.

31. initialize_faults$fault_init_two

Appendix A

Initialization Today: A Summary

All faults are directed to their normal handlers.

32. init_branches

Branches are created for all segments loaded in collections one and two which need them.

33. delete_segs$temp

Delete all unneeded collection two segments.

34. load_system

Read the segments of collection three into the hierarchy.

35. tape_reader$final

Rewind the MST. Shutdown the collection two tape reader.

36. tc_init

Initialize traffic control. Create the first processes (the initializer and idle processes).

37. io_init

Initialize the teletypes in the system. Set up the segment "tty_buf".

38. delete_segs$delete_segs_init

Delete all remaining initialization segments.

39. init_proc$multics

Set up the system search rules. Call "system_startup" in ring 1.

Appendix B

Needed Reconfiguration Primitives

This appendix contains a list of reconfiguration operations which seem to be useful.

Add or delete a dn355 communication computer.

Add or delete an IOM (Input Output Multiplexor).

Add or delete a memory (and hence a system controller).

Add or delete a CPU.

Add or delete a paging device.

Add or delete records from the usable record pool of the paging device.

Add or delete disk storage devices.

Add or delete teletypes.

Add or delete magnetic tapes.

Add or delete any other devices (such as line printers, card readers and card punches).

Turn disk metering on and off.

Allow the size of usable space on disks to grow and shrink (repartitioning).

Change the interrupt cell assignments for the six process interrupts.

Change the number of AST entries of each size.

Expand and contract the size of the AST trailer segment.

Change the traffic control parameters dynamically.

Appendix B

Needed Reconfiguration Primitives

Change the size of the traffic control tables (APT, ITT, DST).

Change the size of the segment "tty_buf".

Change system debugging options.

Change the size of secondary storage device overflow thresholds.

Turn on and off system tracing.

Change the size of the system trace buffer.

Turn operators console logging on and off.