Published: 04/28/67

Identification

Overview of the Multics Initializer
A. Bensoussan

Purpose

The Multics Initializer (MI) is given control by the Bootstrap Initializer (BL.4); its main purpose is to load and initialize the hardcore supervisor and to create the first Multics process. The Multics Initializer then transfers control to the Multics System Control in which the establishment of Multics operation is completed by creating various system service processes.

The basic functions that have to be performed by the MI have been listed in BL.0. As was said in BL.0, the different functions of the MI are not executed in the order they are described there. A different approach has been taken which is explained and justified in this section. Then a general description of each data base and component of the Multics initializer is provided.

Strategy of the Multics Initializer

It might appear possible to perform the various functions of the MI in such a way that each of them would be independent of the other. The MI would be a simple GE645 program, from the beginning to the end, initializing one thing after the other, until all functions are done, and saying at the very last moment: "Now I am a Multics process and I can use all the Multics facilities available to any process". This method has 2 disadvantages:

1. It would require a large amount of special purpose code to perform functions that can be done automatically by the hardcore supervisor modules provided that they be called after their needed environment has been established.

2. It would require a large amount of core memory since all the hardcore supervisor segments that have to be only loaded or active when Multics is operating, would have to be in core at the same time during the execution of the MI. Because of the actual size of the hardcore supervisor, the above consideration is a sufficient reason for not taking this approach.

Therefore, the following method has been taken: The MI starts as a normal GE645 program; this program can be regarded as an "under-developed" Multics process that has a primitive environment, as far as the fault-interrupt handlers and I/O capabilities are concerned. But this program has the following characteristic: It reads the information recorded on the MST in several steps; each time it reads a set of information, it applies to it the appropriate transformation to change the tape information into a Multics facility; then it decides that this facility belongs to it and uses it, so increasing its power little by little, until it has established the environment expected by any Multics process.

The MI contains a main program which issues calls to various entries in Multics initializer's modules. The function performed by each entry and the order in which they are called are such that each of them makes available to the MI some Multics procedures that can be used by the next entries as if the MI were a process, even if it is not yet.

In particular, the problem of core memory space limitations encountered in the first method is solved by making available "page control" and "core control" as soon as possible and by using them to load the rest of the hardcore supervisor, even if "segment control" is not available yet (Initializing also segment control without using the paging mechanism would require a larger amount of core).

Basically, the idea is the following: The Multics missing page fault handler can operate without the help of segment control provided that the page belongs to an active segment; that is, the page table word which causes the fault contains a pointer to the Active Segment Table (AST) entry (or to the Descriptor Segment Table (DST) entry if the page belongs to a descriptor segment).

Therefore, after having initialized the secondary storage used by the file system, the DIM's, the GIM and the interrupt handling mechanism, the Multics Initializer loads and pre-links all the wired-down supervisor segments. Then, an AST entry is created for each existing segment which is neither a wired-down supervisor segment nor a descriptor segment, and a DST entry is created for the MI's descriptor segment; a pointer to the AST or DST entry is placed in the Segment Loading Table (SLT) entry. Finally the core

map is initialized. The interim fault interceptor is told to direct missing page faults to the Multics page fault handler, and missing segment faults to a special purpose segment fault handler (interim2_segfault), the role of which is explained below.

From this point on, the remainder of the hardcore supervisor can be loaded in the "virtual memory" provided by the file system, using the following mechanism: When a segment is to be loaded, at the first reference, a missing segment fault occurs; control is given to the interim2_segfault handler mentioned above. This handler, looking in the SLT, determines if the segment is a normal segment or a descriptor segment; it builds an AST (or DST) entry for this segment, using the information in the SLT, and places a pointer to the created AST (or DST) entry in the SLT entry. Then it calls a primitive in page control to get a hyperpage for the page table (the hyperpage size is found in the SLT entry), it places a pointer to the AST (or DST) entry in each page table word, manufactures a segment descriptor word using the SLT entry information and returns. When a page of the segment is referenced for the first time, a missing page fault occurs, that can be handled properly by the Multics page fault handler. If sometime later, the segment is unloaded and then referenced again, the interim2_segfault handler builds a new page table and the AST (or DST) pointer is copied from the SLT entry to each page table word.

It should be noted that only page retrieval is possible at this point; segment retrieval is not, since none of the segments that have been loaded from the MST has a branch in the file system hierarchy yet. This implies that, before the Multics segment fault handler is available, none of the loaded segments can be deactivated. This is guaranteed by setting the necessary "hold-switches" in the AST entries.

The other advantage of this strategy is to be able to use some of the hardcore supervisor modules to perform some of the initialization functions. The best example that can be given is the example of page and core control that are used by the Multics Initializer as soon as they are available, as explained above. Other examples can be given: The GIM and the interrupt interceptor are initialized and are added to the facilities available to the MI for drum, disc and tape I/O; like page control, segment control will be initialized and used by the MI; branches can be

created in the file system hierarchy using file system routines; the traffic controller routine "create_KPT" is called by the MI to create an entry for itself in the Known Process Table, the process creation module is called by the MI in order to manufacture the "stack history" which is a process creation data base, etc.

A fundamental remark has to be made as far as the behavior of the process exchange during initialization is concerned: The MI uses the file system routines to read from and write on secondary storage. The file system, in turn uses the Multics mechanism to wait for an I/O operation. Thus entries "block" and "wakeup" in the process exchange will be exercised. It is obvious that if the MI calls block before the traffic controller is initialized, or if the drum interrupt handler pretends to send a wakeup to the file system drum manager process before it is created, one can expect the worst catastrophies to occur.

Therefore, modules "block" and "wakeup" are disabled during the most part of MI; that is, they merely perform a return. They will be enabled at the appropriate point of the MI when the traffic controller initialization is completed.

The Multics Initializer's strategy is implemented in 4 functional parts:

1. The first part, referred to as "Part 1", makes known the hardware configuration, loads a few modules of the hardcore supervisor (mainly the GIM, the interrupt interceptor, the interrupt handlers and the process exchange) needed for I/O operations, initializes the secondary storage devices if the file system hierarchy has been destroyed, and initializes a DIM for each secondary storage device available to the file system.

2. The main purpose of the second part, referred to as "Part 2", is to load the rest of the wired-down segments and initialize page control and core control. Part 2 is the most critical part of the Multics Initialization in the sense that it requires a large amount of core memory. Therefore, after Part 1, all the core filled by initialization routines that are no longer used is returned and made available for Part 2.

3. The third part, referred to as "Part 3", is concerned with initialization of segment control. The rest of the hardcore supervisor, that need not be wired down, is loaded in "virtual memory" made available in Part 2.

4. In the last part, referred to as Part 4, the rest of the I/O system, the fault interceptor, and the traffic controller are initialized. Block and wakeup are enabled and the Multics initializer ends by calling the Multics system control procedure.

Various components of the MI

1. Data bases. The following data bases are used by the Multics initializer during the whole initialization.

   a. The Multics system tape (BL.1)

   b. The Segment Loading table (BL.2)

   c. The descriptor segment (Described in BL.4.01)

   d. The system configuration table (BL.3)

2. Modules. The Multics initializer contains the following modules:

   a. Initializer control module (BL.5.01)

   b. Segment loader module (BL.6)

   c. Initialization linkage module (BL.7)

   d. I/O system initializer module (BL.8)

   e. Fault-interrupt initializer module (BL.9)

   f. File system initializer module (BL.10)

   g. Traffic controller initializer module (BL.11)

   h. System configuration table generator module (BL.3)

   i. Interim fault interceptor module (BL.5.02)

Each of these modules may consist of one or several segments.

A brief description of each data base and each module is given below, to get the reader familiar with them, so that he will not need to perform a dynamic linkage each time an inter MSPM section reference is made in the rest of the BL sections.

Multics system tape (BL.1.01)

The Multics system tape (MST) contains all information that needs to be loaded during system initialization, that is, configuration segments, supervisor segments, initialization segments and all segments needed by the hierarchy reconstruction process. As a consequence of the strategy explained above, the MST is read at several distinct times; therefore it is organized into "collections". Each time a set of segments is to be loaded, 2 collections are involved: the "loadlists" collection followed by the "library" collection. The loadlists collection is made up of several loadlists; each of them is a segment and contains a group of segment names. Once a given loadlist has been selected, it defines a specific set of segments to be loaded from library collection.

In the overview BL.0 it was explained that with a given configuration CONF(i) was associated a loadlist CONF.LL(i), in the configuration loadlists collection, naming every segment CONF.SEG(i,k) that had to be loaded from the configuration library collection. Because of the strategy taken in the MI, the configuration information is broken up into 2 parts which appear at two different places in the MST; the first part is loaded at the very beginning of the MI, while the loading of the rest is postponed until page control is available.

Therefore, the statement made in the overview BL.0 mentioned above should be expanded as follows: with a given configuration CONF(i) are associated:

a. a loadlist CONF.LL.1(i) in the configuration part 1 loadlists collection, naming every segment CONF.SEG.1(i,k) that has to be loaded from the configuration part 1 library collection.

b. a loadlist CONF.LL.2(i) in the configuration part 2 loadlists collection, naming every segment CONF.SEG.2(i,k) that has to be loaded from the configuration part 2 library collection.

The same remark is to be made with respect to the supervisor segments. The supervisor segments are not loaded in one step but in three. With a given version SUP(j) of the supervisor are associated:

a. a loadlist SUP.LL.1(i) in the supervisor part 1 loadlist collection, naming every segment SUP.SEG.1(j,k) that has to be loaded from the supervisor part 1 library collection.

b. Same as (a) with "2" instead of "1".

c. Same as (a) with "3" instead of "1".

Segment loading table (BL.2.01)

The segment loading table (SLT) is a segment consisting of one entry for each segment created or loaded during the system initialization. Entries are of fixed length and are indexable by segment number. Each time a segment is loaded from the MST an entry is created in the SLT, defining the segment name(s), the path-name, the maximum length, the current length, the access rights, etc. The information needed by the MI to manufacture an entry in the SLT is recorded on the MST in a "header" which precedes the segment.

The SLT is used intensively by the Multics Initializer for various purposes: When a segment is loaded, the information needed to build the segment descriptor word and the page table is found in the SLT; when a linkage fault occurs, the SLT is used to find out the segment numbers of the referenced segment and its associated linkage section; during the file system initialization the status of the segment (wired-down, loaded, active, normal) and the path name used to create a branch in the hierarchy, are drawn from the SLT; information needed to combine the hardcore supervisor linkage section segments is taken by the "pre-linker" from the SLT, etc.

Descriptor segment

The descriptor segment used by the MI is paged; as described in BL.0, hardcore supervisor segments are assigned segment numbers from 0 to n-1 while initialization segments are given segment numbers from n to 2n-1. The value chosen for n is 2048. When a segment is to be loaded from the MST, the "header" is first read, and the SLT entry is built, thus assigning a segment number to the segment. Then the MI reads the segment from the MST and moves it, word by word, into the segment. When the first word is requested to be moved into the segment, a missing segment fault occurs; control goes to the interim segfault handler which builds the segment descriptor word at the appropriate location within the descriptor segment and with the appropriate access right bits, using the SLT entry.

System configuration tables (BL.3.01)

The system configuration tables are made up of three tables, each of them being a separate segment:

a. Major configuration table (MCT), containing information that is concerned with memory controllers and active devices.

b. File system configuration table (FSCT), containing information pertaining to each secondary storage device accessible to the basic file system.

c. Device configuration table (DCT), containing information pertaining to each I/O device used by the I/O system.

Since the DCT is a very large segment and since only a small part of it is needed to initialize page control, the following attitude is taken with respect to the system configuration tables: The MCT, the FSTC and the portion of the DCT that is needed to initialize page control are manufactured in Part 1 of the MI; the rest of the DCT will be manufactured in Part 3, the virtual memory being available at this time.

MCT, FSCT and DCT are manufactured by the system configuration table generator, which is one of the Multics initializer's modules. MCT, FSCT and the first portion of DCT are built using the configuration segments loaded in Part 1; the rest of the DCT is built using the configuration segments loaded in Part 3.

Initializer control module (BL.5.01)

The Initializer control module contains only one segment, the name of which is "initializer_control". It consists of a sequence of calls to the other Multics initializer modules and contains the whole logic of the MI; the modules that it calls and the order in which they are called is such that, after each call (or group of calls) a new Multics facility is made available, which expands the MI's environment until it becomes a normal process. The initializer control is called by the bootstrap initializer and calls in turn, the Multics system procedure at the end of the Multics initializer.

Segment loader module (BL.5)

This module contains the following segments:

segment_loader	(BL.5.01)
tape_reader	(BL.5.02)

interim1_pagefault	(BL.5.03)
interim1_segfault	(BL.5.03)
core_manager	(BL.5.03)
slt_manager	(BL.2.02)

1. Segment_loader. It is called by the initializer control program when segments have to be loaded from the MST.

2. Tape_reader. It contains 2 entries:

   a. tape reader$tape_reader is called by the segment loader which requests to move a given number of logical words into core starting at a given location. This tape reader is responsible for issuing connects GIOC while the GIM is not initialized.. When the GIM is ready, the tape reader has to use it to issue a connect.

   b. tape_reader$use_gim is called by the initializer control program to tell the tape reader that the GIM is initialized and has to be used for connect operations.

3. interim1_pagefault. It is called by the interim fault interceptor when a missing page fault occurs; it assigns a hyperpage to the segment in which the page fault occurred. It will be used until the Multics missing page fault handler is available, at the end of Part 2.

4. interim1_segfault. It is called by the interim fault interceptor when a missing segment fault occurs. It assigns a page table and a segment descriptor word to the missing segment. It will be used until the interim2 segment fault handler (which is compatible with the Multics page fault handler) is provided, at the end of Part 2.

5. Core_manager. It contains 3 entries and the interim core map.

   a. Core_manager $ assign_core is called by interim1_page fault and interim1_segfault.

b. Core_manager $ update_core_map is called by the initializer control program to tell the core manager that the configuration is known and that the core map has to be revised accordingly.

c. Core_manager $ free_core is called by the initializer control program at the end of Part 1 to return the core occupied by all initialization segments that are no longer needed.

d. Core_manager $ core_map

6. slt_manager. It is a utility routine used by the segment_loader to maintain the SLT, and used also by other procedures to get information from the SLT.

Initialization linkage module (BL.7)

It contains 3 segments:

linker (BL.7.01)

pre_linker (BL.7.02)

datmk_ (BL.7.03)

1. Linker. This segment is used for 2 purposes:

a. To establish the dynamic linkage between the various Multics initializer segments. When used for this purpose it is called by the interim fault interceptor.

b. To change linkage faults into correct machine addresses, in the hardcore supervisor linkage sections. When used for this purpose it is called by the pre-linker.

This linker is not the Multics linker because it has to request the segment number of the reference segment from the SLT manager. In BL sections, this linker is referred to as the "initialization linker".

2. Pre-linker. This segment is called by the initializer control program to combine and prelink the hardcore supervisor linkage sections. It calls the initialization linker.

3. Datmk_. It is a segment grower procedure used in the implementation of PL/I static storage. It is called by the initialization linker when it recognizes a "trap before link". If the referenced segment does not exist, datmk creates it; if the linkage section of the referenced segment does not exist, datmk creates a linkage section; if the referenced symbol is not in the linkage section, datmk places the external symbol definition in the linkage section. Then it returns to the initialization linker.

I/O system initializer module (BL.8)

This module consists of 2 segments:

io_init_1 (BL.8.01)

io_init_2 (BL.8.02)

1. io_init_1 is called by the initializer control program during Part 1 of MI. It initializes the GIM and its data bases so that the GIM can accept calls by the file system initializer to write on disc, and calls by the tape reader to read the Multics system tape.

2. io_init_2 is called by the initializer control program during Part 4 of MI. It initializes the tape controller interface module (TCIM) and its data bases.

Fault-interrupt initializer module (BL.9)

It contains 2 segments:

fault_init (BL.9.01)

interrupt_init (BL.9.02)

1. fault_init has two entries called by the initializer control program during Part 4 of the MI.

   a. fault_init $ one initializes the fault interceptor

   b. fault_init $ two initializes the fault vector so that it transfers control to the fault interceptor when a fault occurs; that is, this call operates the switching from the interim fault interceptor to the Multics fault interceptor.

2. interrupt_init has two entries called by the initializer control program.

   a. interrupt_init $ one is called in Part 1 and initializes the interrupt interceptor

   b. interrupt_init $ two is called in Part 1 and initializes the interrupt vector in its final form so that, upon completion of an I/O operation for the MST, the drum or the discs, the interrupt interceptor will be entered.

File system initializer module (BL.10)

It contains 5 segments that are called by the initializer control program:

fs_init_1 (BL.10.01)

fs_init_2 (BL.10.02)

fs_init_3 (BL.10.03)

fs_init_4 (BL.10.04)

fs_windup (BL.10.04)

1. fs_init_1 is called in Part 1 after the GIM and the interrupt interceptor have been initialized. It initializes all secondary storage devices used by the file system, if the hierarchy must be reloaded. It also initializes some of the data bases used by the file system: Device disposition table (DDT), empty active segment table (AST), empty descriptor segment table (DST), empty process segment table (PST), empty core map, empty wired-down process waiting table, I/O queues and all device interface modules (DIM).

2. fs_init_2 is called in Part 2, after all the wired-down supervisor segments have been loaded and pre-linked. It initializes page control, core control and core map. Furthermore it provides the MI with an interim segment fault handler (interim2_segfault) which performs the functions required by page control to be able to work properly while segment control is not available yet.

   Before segment control is available, all existing segments must be kept "active".

3. fs_init_3 is called in Part 3 after all the hardcore supervisor modules have been loaded and pre-linked. It creates a branch in the file system hierarchy for each existing segment (hardcore supervisor and initialization segments), initializes segment control and the rest of the basic file system.

4. fs_init_4 is called in Part 4 after the interim fault interceptor has been told, by the initializer control program, to call the Multics segment fault handler when a missing segment fault occurs. A certain number of segments were kept "active" just to allow page control to work without segment control. Now, the Multics missing segment fault handler is operational; therefore fs_init_4 reduces the AST entry-hold count by one for each segment whose status is "normal".

   Then, if the hierarchy must be reloaded, fs_init_4 loads from the MST all the segments needed by the hierarchy reconstruction process.

5. fs_windup is called in Part 4 before the MI ends by calling Multics system control. fs_windup returns all core which is currently wired-down in the behalf of the Multics initializer.

Traffic controller initializer module (BL.11)

The traffic controller initializer is made of 1 segment tc_init. It is called by the initializer control program in Part 4.

The procedure tc_init creates, for the MI, the per-process information used by the traffic controller: an entry in the known process table (KPT), an entry in the active process table (APT), a process data block; then it creates the system data bases used by process creation: template descriptor segment, stack history; it creates the necessary system processes: file system device monitor process and one idle process for each processor. Then it enables block and wakeup.

System configuration table generator module (BL.3.02)

It contains 3 segments:

mct_generator

fset_generator

dct_generator

They are responsible for translating the configuration segments (loaded from the configuration library collections) from symbolic form to binary form, in the formats required by the major configuration table (MCT), the file system configuration table (FSCT) and the device configuration table (DCT).

Interim fault interceptor module (BL.5.02)

It consists of 1 segment with three entries

```
interim_fi $ interim_fi

interim_fi $ use_mode_2

interim_fi $ use_mode_3
```

The interim fault interceptor is given control when a missing page fault, missing segment fault and a linkage fault occur. On a linkage fault, the interim fault interceptor always calls the initialization linker; on the other hand, on a missing page or segment fault, it may have to call different segments depending on where the initialization stands. That is why it runs under 3 different modes:

Mode 1. the handlers are: interim1_pagefault and interim1_segfault

Mode 2. the handlers are: (multics) pagefault and interim2_segfault

Mode 3. the handlers are: (multics) pagefault and (multics) segfault

At the beginning of the MI, the interim fault interceptor runs under Mode 1; when page control is initialized, the initializer control program calls the entry interim_fi $ use_mode_2, which causes the interim fi to switch to mode 2; where segment control is initialized, a call from the initializer control program to interim_fi $used_mode_3 causes the interim fi to switch to mode 3.

Initial environment of the MI

The Multics initializer is given control by the bootstrap initializer by:

```
call <initializer_control>|[initializer_control]
```

The initial environment of the MI is as follows:

1. All the Multics initializer's modules are not in core when this call is issued. Only a few of them have been loaded and initialized by the bootstrap initializer, so that the Multics initializer can run properly. These segments are (see Figure 1):

   descriptor segment

   initializer_control

   segment_loader

   tape_reader

   mailbox

   physical_record_buffer

   slt_manager

   slt

   fault_vector

   interim_fi

   interim1_pagefault

   interim1_segfault

   core_manager

   linker (initialization linker)

   pds (process data segment)

   stack_0

   stop

   Other segments will be loaded by the segment loader when requested by the initializer control program to do so.

2. The interim fault interceptor runs under mode 1 (see BL.5.02), that is: when a missing page fault occurs, interim1_pagefault allocates a hyperpage; when a missing segment fault occurs, interim1_segfault builds a segment descriptor word and a page table for the missing segment; when a linkage fault occurs, the initialization linker changes the fault pair into a correct machine address; timer runout fault and connect fault are ignored (the interim fault interceptor merely restores the control unit).

3. Interrupts coming from the bootload GIOC through status channels 0 to 11 are directed to <tape_reader>|[interrupt].

4. All other faults or interrupts are directed to the segment stop, which causes the initialization to stop.

5. The physical record buffer is initialized in such a way that the next logical word of the MST that will be delivered by the tape reader is the first word of collection 2 (collection 2 contains the loadlists for the first part of the configuration).

6. The SLT contains one entry for each existing segment.

7. All the calls use the hardcore stack.

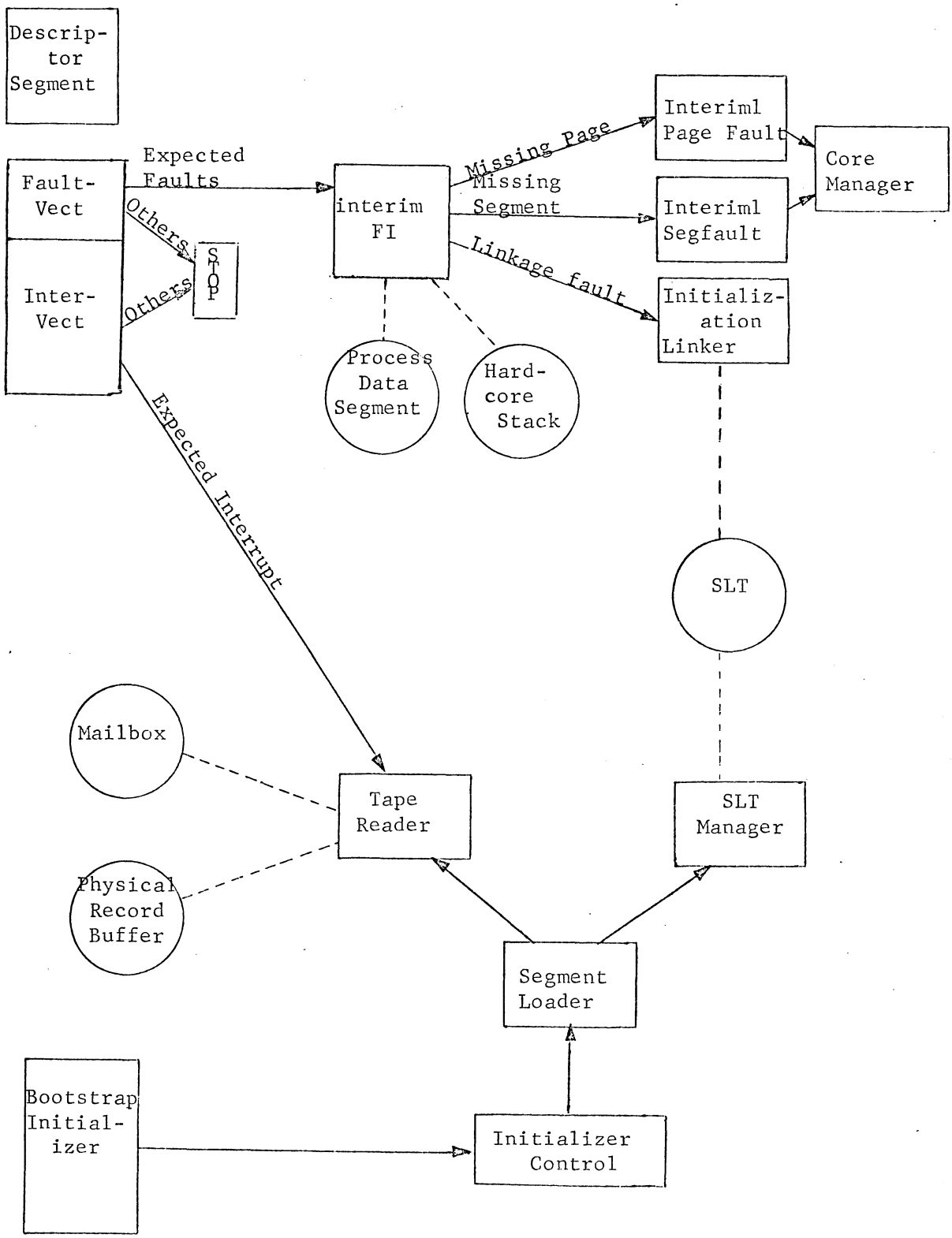

Figure 1: Initial environment of Multics Initializer