Published: 05/1/67

Identification

Initializer Control Program - Initial version
A. Bensoussan

Purpose

The initializer control program is one of the Multics initializer's components. It constitutes the logical skeleton of the Multics initializer, since it consists of a sequence of calls to the appropriate initialization routine, at the appropriate time.

This section describes the initializer control program that will be used in Phase I and it will be superseded by a complete description.

Phase I restrictions

1. The Multics initializer does not interact with the operator.

2. There is only one possible version of the hardware configuration and one possible version of the supervisor; therefore, the segment loader does not have to select the appropriate loadlist in a loadlist collection.

3. The system configuration table is made up of 3 segments:

   a. Major configuration table (MCT)
   b. File system configuration table (FSCT)
   c. Device configuration table (DCT)

   In the MST, the MCT and the FSCT are recorded in the final binary form; thus they do not need to be translated from symbolic form to binary form. On the other hand, the DCT is recorded in symbolic form and has to be translated in binary form.

   In Phase I, the system configuration table generator is not implemented. The DCT is manufactured from the symbolic segments by the io_initializer.

4. Since there is only one process, the process exchange does not need to be initialized. As a consequence "block" and "wakeup" will never be enabled.

5. The protection mechanism is not implemented in Phase I. The notion of the ring still exists but a ring merely represents a set of segments, without any protection, that is, all rings have the same number. This conception of the particular ring structure used in phase I is preferable to the one which would consist of having only one ring, because it allows us, in the documentation, to refer to the "hardcore supervisor" segments and "outside hardcore supervisor" segments.

6. The signal mechanism is not implemented in Phase I. The fault interceptor calls directly the appropriate fault handler instead of calling the signal routine.

7. In the full Multics, when a process is created, the linker and its utility routines to be used by the process have to be pre-linked by the process creation module.

   In Phase I, these segments are loaded from the Multics system tape and pre-linked as if they were part of the hardcore supervisor.

Discussion

The initializer control program is loaded and called by the bootstrap initializer. The calling sequence is:

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

The environment available to it when it is called is described in detail in BL.5.00.

The initializer control program contains 4 parts (all of them in same segment) corresponding to the 4 logical parts of the Multics initializer (BL.5.00).

This section contains the following paragraphs:

a. Initializer control program
b. Description of part 1
c. Description of part 2
d. Description of part 3
e. Description of part 4

In (a), the complete list of all calls issued from the initializer control program is given; (b) (c) (d) and (e) provide a short description of what each call is supposed to perform in each part, and indicate the MSPM sections where a complete description can be found.

Initializer control program

Part 1:

```
 1.  Call segment_loader                      /* load Part 1
                                                 configuration      */
 2.  Call core_manager $ update_core_map      /* update core map    */
 3.  Call segment_loader                      /* load Part 1
                                                 supervisor         */
 4.  Call pre_linker $ combine_set            /* prelink Part 1
                                                 supervisor         */
 5.  Call io_init_1                           /* initialize GIM     */
 6.  Call tape_reader $ use_gim               /* tell tape reader
                                                 GIM ready          */
 7.  Call interrupt_init $ one                /* initialize interrupt
                                                 interceptor        */
 8.  Call interrupt_init $ two                /* initialize
                                                 interrupt vector   */
 9.  Call fs_init_1                           /* secondary storage
                                                 devices            */
10.  Call core_manager $ free_core            /* get core for
                                                 Part 2             */
```

Part 2:

```
 1.  Call segment_loader                      /* load Part 2
                                                 supervisor         */
 2.  Call pre_linker $ combine_set            /* prelink Part 2
                                                 supervisor         */
 3.  Call fs_init_2                           /* initialize page
                                                 fault              */
 4.  Call fault_init $ one                    /* initialize fault
                                                 interceptor        */
 5.  Call interim_fi $ use_mode_2             /* switch interim_fi
                                                 to mode 2          */
```

Part 3:

```
1.  Call segment_loader                 /* load Part 3
                                           supervisor                */
2.  Call segment_loader                 /* load Part 2
                                           configuration             */
3.  Call pre_linker $ combine_set       /* prelink Part 3
                                           supervisor                */
4.  Call fs_init_3                      /* initialize segment
                                           fault                     */
5.  Call interim_fi $ use_mode_3        /* switch interim_fi
                                           to mode 3                 */
```

Part 4:

```
1.  Call fs_init_4                      /* reload file system
                                           hierarchy                 */
2.  Call pre_linker $ redirect          /* references out of
                                           hardcore                  */
3.  Call io_init_2                      /* initialize TCIM           */
4.  Call fault_init $ two               /* initialize fault
                                           vector                    */
5.  Call tc_init                        /* initialize traffic
                                           controller                */
6.  Call fs-windup                      /* delete initialization
                                           segments                  */
7.  Call multics $ system_control       /* enter multics system
                                           control                   */
8.  Call shut_down                      /* shut down the system      */
```

Description of Part 1

The main purposes of Part 1 are to:

a. Initialize the first part of the system configuration table.

b. Load the first part of wired-down hardcore supervisor and initialization segments needed in Part 1.

c. Combine linkage sections and perform the pre-linkage of hardcore supervisor segments that have been loaded.

d. Initialize the GIOC interface module (GIM) and its data bases.

e. Initialize all secondary storage devices used by the file system.

The following calls are issued from the initializer control program:

1. Call segment_loader (BL.6.01)

   When invoked at this point the segment loader reads the Part 1 configuration loadlist (collection 2). This loadlist contains the names of all the segments that have to be read from the Part 1 configuration library (collection 3).

   Then, the segment loader reads the segments mentioned in the loadlist from collection 3.

   In Phase I, the configuration segments read in Part 1 are:

   a. The major configuration table (MCT) itself in binary form

   b. The file system configuration table (FSCT) itself in binary form

   c. The symbolic configuration segments containing the necessary information to build the portion of the DCT needed in Part 1. These segments in symbolic form will be translated to binary form and stored in the DCT, by a call to io_init_1.

2. Call core_manager $ update_core_map (BL.6.03)

   Up to this point, the core manager was assuming that a certain amount of core was available immediately following the base address of the GIOC used in the bootload. Now it can update its core map using the major configuration table (MCT) that has been read by the segment loader.

3. Call segment loader (BL.6.01)

   When invoked at this point, the segment loader reads the Part 1 supervisor loadlist (collection 4). This loadlist contains the names of all supervisor and initialization segments that have to be loaded from the Part 1 supervisor library (collection 5).

Then the segment loader reads the segments mentioned in the loadlist from collection 5. Among them are the portion of the wired-down hardcore supervisor segments needed in Part 1.

4. Call pre_linker $ combine_set (BL.7.02)

For all the hardcore supervisor segments that have been loaded and that have a combinable linkage section, the linkage section information is combined in the appropriate special linkage segment (wired_hcs.link, loaded_hcs.link or active_hcs.link).

Then, for all external references made in any hardcore supervisor segment that is in core, to any segment that is also in core, the link pairs in the hardcore supervisor linkage sections are changed from faults to correct machine addresses.

5. Call io_init_1 (BL.8)

The GIOC interface module (GIM) and its data bases are initialized in such a way that the GIM can accept calls from the file system initializer for reading and writing on discs, and also from the Multics initializer's tape reader for reading the MST.

The actions taken by io_init_1 are the following:

a. Initialize device configuration table (DCT).

The DCT table is one segment of the system configuration table. It is a very large segment; therefore, only the portion needed in Part 1 is built by io_init_1; the rest of it will be built in io_init_2. The necessary segments to manufacture the first portion of the DCT have been loaded with Part 1 configuration segments; they are in symbolic form; they are translated by io_init_1 into binary form and stored in the DCT.

b. Initialize GIM and its data bases. The data bases are the logical channel table (LCT), the channel assignment table (CAT), the channel status table (CST) and the wired down storage area.

Upon return from this call, the GIM is ready to accept I/O requests from the Multics initializer for tape and discs.

6. Call tape_reader $ use_gim (BL.6.02)

   Before the GIM was initialized, the tape reader was creating by itself the connect control words for reading the tape. Now it has to request connect operations from the GIM. This call to the tape reader is merely to inform it that the GIM is initialized and has to be used.

7. Call interrupt_init $ one (BL.9.02)

   The interrupt interceptor and its data bases are initialized:

   a. The value lp-lb for the interrupt interceptor is obtained and stored "inside" the interrupt interceptor procedure.

   b. Pointers to various locations of the processor and concealed stacks are built and stored "inside" the interrupt interceptor procedure.

   c. The processor stack is initialized, that is, the 4 pointers scu_ptr, stb_ptr, streg_ptr and next_sp_ptr and set.

   d. The mask table in the system communication segment, and the interrupt decode table in the interrupt interceptor itself are initialized.

8. Call interrupt_init $ two (BL.9.02)

   The interrupt vector is initialized to its final form.

   From this point on, tape, disc and drum I/O can take place using the Multics interrupt handling mechanism except that the modules "wakeup" and "block" in the process exchange are disabled (they execute a return) until the coexistence of at least 2 processes, situation that will occur during the traffic controller initialization, in Part 4.

   For Phase I, traffic controller initialization is not needed; therefore, wakeup and block will never be enabled.

9. Call fs_init_1 (BL.10.02)

   The major functions performed by this call to the file system initializer are:

   a. Initialize the hierarchy by preparing free storage map in each secondary storage device and creating an empty root directory.

      This action is taken only if the file system hierarchy has been destroyed or must be reloaded.

   b. Define the areas of storage to be used by the version of Multics currently being initialized.

   c. Initialize the device disposition table (DDT).

   d. Initialize the system segment tables (SST): the active segment table (AST), the descriptor segment table (DST) and the process segment table (PST) are initialized to appear empty.

   e. Initialize the core map in such a way that each block of core available to this version of Multics appears currently unassigned.

   f. Initialize the wired-down process waiting table (PWT) to appear empty.

   g. Initialize the I/O queues common to all DIM's; then initialize a DIM for each secondary storage device available to the file system.

10. Call core_manager $ free_core (BL.6.03)

    Part 2 of the Multics initializer is the most critical part as far as the core requirement is concerned.

    For this reason, at the end of part 1, all the segments which are no longer needed are deleted and the core map maintained by the core manager is updated.

The information indicating that a segment can be deleted at this point is found in the corresponding SLT entry.

The segments deleted now are:

- Bootstrap 1
- Bootstrap 2
- io_init_1
- fs_init_1 and all the initialization segments that it calls, and their associated linkage sections.

Description of Part 2

The main purposes of Part 2 are to:

a. Load the rest of the wired-down supervisor segments and the portion of the Multics initializer segments needed in Part 2.

b. Combine linkage sections and perform the prelinkage of hardcore supervisor segments that have been loaded.

c. Make Multics missing page fault handling available.

d. Make interim2 missing segment fault handling available.

After Part 2 is completed, the remaining segments of the hardcore supervisor can be loaded in virtual memory, being eventually moved into secondary storage by the usual Multics paging algorithm.

The following calls are issued from the initializer control program:

1. Call segment_loader (BL.6.01)

   When invoked at this point, the segment loader reads the Part 2 supervisor loadlist (collection 6). This loadlist contains the names of all supervisor and initialization segments that have to be loaded from the Part 2 supervisor library (collection 7).

Then the segment loader reads the segments mentioned in the loadlist from collection 7. Among them are the rest of the wired-down hardcore supervisor segments.

At this point, all the wired-down segments are in core.

2. Call pre_linker $ combine_set (BL.7.02)

For all the hardcore supervisor segments that have been loaded in Part 2 and that have a "must-be-combined" linkage section, the linkage section information is combined in the appropriate special linkage segment (wired_hcs.link, loaded_hcs.link or active_hcs.link).

Then, for all external references made in any hardcore supervisor segment that is in core, to any segment that is also in core, the link pairs in the hardcore supervisor linkage sections are changed from faults to correct machine addresses.

3. Call fs_init_2 (BL.10.03)

The following actions are taken by the file system initializer:

a. Set the descriptor words for the segments "hardcore_ds" and "current_ds" to point to the Multics initializer's descriptor segment.

b. Create an entry for the hardcore descriptor segment in the descriptor segment table (DST).

c. Create an entry for the Multics initializer in the process segment table (PST) and link it to the DST entry.

d. Create an entry in the active segment table (AST) for each segment of the SLT (except for wired-down hardcore supervisor segments and for the descriptor segment) and place, in the SLT entry, a pointer to the corresponding AST entry.

e. Set every page table word in such a way that: for an existing page, it contains a "written-bit" ON, for a non-existing page, it contains a pointer to AST entry of the segment to which the page belongs (or a pointer to the DST entry if the segment is the descriptor segment).

f. For each segment which is neither a "wired-down" nor a "loaded" hardcore supervisor segment, attach to the corresponding AST a process trailer showing that the segment is used by the Multics initializer.

g. Update the core map.

Upon return from this call, the Multics missing page fault handler and an interim segment fault handler are operable.

The name of this interim handler is "interim2_segfault". Missing segment faults cannot be handled by the Multics handler because none of the existing segments has a branch in the hierarchy. Branches will be established in Part 3, where the Multics missing segment handler will be made available to the Multics Initializer.

Interim1_segfault, when invoked, used to build a segment descriptor word and a page table for the missing segment, taking the needed information in the SLT and requesting memory from the core manager.

Interim2_segfault still performs these 2 functions, of course, but it does it in such a way that the page fault handler made available in Part 2 can run properly during Part 3, while the Multics missing segment is not available:

a. If no pointer to the AST is found in the SLT, an AST entry is created and a pointer to the AST entry is placed in the ALT entry.

b. A call is made to page control to provide a page table. Each page table word contains a pointer to the AST, which allows page fault to work.

c. A descriptor segment word is manufactured.

Furthermore, in Part 3 during a short period of time (while the file system initializer manipulates directory segments), missing segment faults will occur for segments that are not in the SLT. These faults will be directed to interim2_segfault by the interim fault interceptor; but interim2_segfault cannot accomplish its job without SLT entry. In this case, interim2_segfault passes the fault to the Multics missing segment fault handler, which is ready to handle this type of fault.

During execution of Part 3, none of the existing segments can be deactivated. This is guaranteed by setting the necessary "hold switches" in the AST entry.

4. Call fault_init $ one (BL.9.01)

The fault interceptor and its data bases are initialized. The fault interceptor will be switched to only in Part 4; the reason why this call is issued in Part 2 is that the processor communication tables will be needed in Part 3 by the connect fault handler (see BL.5.02).

a. The value lp-lb for the fault interceptor is obtained and stored "inside" the fault interceptor procedure.

b. Pointers to various locations of the concealed and the processor stacks are built and stored "inside" the fault interceptor procedure.

c. The processor communication tables are initialized.

d. Entries in the process definition segment that are used by the fault interceptor are initialized.

Note that the concealed stack is not initialized in this call since it has already been initialized by the bootstrap initializer.

5. Call interim_fi $ use_mode_2 (BL.5.02)

   Up to now, the interim fault interceptor was directing missing page faults and missing segment faults to the segments "interim1_pagefault" and "interim1_segfault" respectively.

   Upon return from this call, it directs them to the segments "pagefault" and "interim2_segfault" respectively.

   The standard Multics dynamic memory allocation mechanism is now operational.

Description of Part 3

The main purposes of Part 3 are to:

a. Initialize the rest of the system configuration table.

b. Load the rest of the hardcore supervisor segments using the Multics missing page fault handling and the interim 2 missing segment fault handling made available in Part 2.

c. Combine linkage sections and perform the prelinkage of hardcore supervisor segments that have been loaded.

d. Establish branches in the hierarchy for segments already loaded from the MST.

e. Make the Multics missing segment fault handling available.

The following calls are issued from the initializer control program:

1. Call segment_loader (BL.6.01)

   When invoked at this point the segment loader reads the Part 3 supervisor loadlist (collection 8). This loadlist contains the names of all the supervisor and initialization segments that have to be loaded from the Part 2 supervisor library (collection 9).

Then the segment loader reads the segments mentioned in the loadlist from collection 9. Among them are the rest of the hardcore supervisor segments, that is all the hardcore supervisor segments that have the status "loaded", "active" or "normal".

At this point, the hardcore supervisor is completely loaded.

2. Call pre_linker $ combine_set (BL.7.02)

For all the hardcore supervisor segments that have been loaded in Part 3 and that have a "must-be-combined" linkage section, the linkage section information is combined in the appropriate special linkage segment (wired_hcs.link, loaded_hcs.link or active_hcs.link).

Then, for all external references made in any hardcore supervisor segment that is in core, to any segment that is also in core, the link pairs in the hardcore supervisor linkage sections are changed from faults to correct machine addresses.

At this point, the hardcore supervisor is completely prelinked, as far as inter-references within the hardcore ring are concerned. References to segments outside of the hardcore ring will be prelinked in Part 4.

3. Call segment_loader (BL.6.01)

When invoked at this point the segment loader reads the second part of the configuration loadlist (collection 10). This loadlist contains the names of all the segments that have to be read from the second part of the configuration library (collection 11).

Then the segment loader reads the segment mentioned in the loadlist from collection 11.

These segments are the segments in symbolic form needed to build the rest of the device configuration table (DCT). They will be translated into binary form and stored in the DCT by a call to io_init_2.

4. Call fs_init_3 (BL.10.04)

This call to the file system initializer establishes branches in the hierarchy for segments loaded from the MST, so that the Multics segment fault handler can operate properly. The following actions are taken:

a. Initialize the file system static storage constants.

b. Initialize the normal process waiting table (PWT) to appear empty.

c. Initialize the known segment table (KST) for the future Multics initializer process.

d. Create an entry in the hardcore segment table (HST) for each hardcore supervisor segment.

e. Create an entry in the active segment table (AST) for the root directory.

f. If the hierarchy must be restored, create 2 directory branches: "system_root" for segments of the Multics initializer, the hardcore supervisor, and the hierarchy reconstruction process; and "Multics_root" for all other segments.

g. Establish branches in the file system hierarchy for all the segments listed in the SLT.

h. Update the HST to include the unique identifiers of the hardcore supervisor segments.

i. For each segment listed in the SLT, that has an AST entry, a KST entry is built, and the AST entry is linked to the AST entry of its parent directory segment.

From this point on, the Multics initializer appears to the file system as an active and loaded process.

5. Call interim_fi $ use_mode_3 (BL.5.02)

Upon return from this call, the interim fault interceptor does not direct missing segment faults

to "interim2_segfault" but to the Multics "segfault" segment, so that this fault is handled by the standard Multics mechanism.

At this point, the Multics missing segment fault handler is operational.

Description of Part 4

The main purposes of Part 4 are to:

a. Complete file system initialization, and reload hierarchy if necessary.

b. Arrange the linkage for references from the hardcore supervisor to segments in other protection rings.

c. Complete I/O system initialization.

d. Initialize the Multics fault interceptor and switch to it.

e. Initialize the traffic controller (not needed in Phase I).

The following calls are issued from the initialized control program:

1. Call fs_init4 (BL.10.04)

   The following actions are taken by the file system initializer:

   a. For each segment listed in the SLT with the status "normal", the AST entry hold count is reduced by 1, and the AST entry is removed if the segment should be deactivated at this time.

   b. If the file system hierarchy must be reloaded, load from the MST all segments required to operate the hierarchy reconstruction process. These segments are in collection 12.

2. Call pre_linker $ redirect (BL.7.02)

   For all references from the hardcore ring to other protection rings, the link pairs are redirected through a special linkage segment "out_hcs.link" that may be referenced later.

   At this point all linkage faults have been eliminated from the hardcore supervisor.

3. Call io_init_2 (BL.8)

   The rest of the device configuration table (DCT) is manufactured. The DCT is part of the system configuration tables. The first portion of the DCT has been initialized in io_init_1. All configuration segments needed to build the rest of the DCT have been loaded with Part 2 configuration segments; they are in symbolic form; they are translated by io_init_2 into binary form and stored in the DCT.

   Then the tape controller interface module (TCIM) and its data bases are initialized.

4. Call fault_init $ two (BL.9.01)

   This call, initializing the fault vector to its final form, causes the Multics Initializer to switch from the interim fault interceptor to the Multics fault interceptor.

5. Call tc_init (BL.11)

   Traffic controller initialization is not needed in Phase I; it is included here as a reminder for later. This call performs the following steps:

   a. Initialize the Known Process Table (KPT) and create an entry for the Multics Initializer.

   b. Initialize the Active Process Table (APT) and create an entry for the Multics Initializer.

   c. Initialize the process data block with information that belongs to the Mutlics Initializer.

d. Initialize the processor data block with process id of the Multics Initializer and with information that belongs to the running processor.

e. Initialize data bases of the process creation module.

f. Create necessary system processes specified in the system configuration table (in particular, the file system device monitor process is created).

g. Enable the functions "Block"and "Wakeup".

h. Initialize all other processors by creating an idle process for each of them.

Upon return from this call the Multics Initializer is a normal Multics process.

6. Call fs_windup (BL.10.04)

This call is made to the file system initializer to return all core which was required to be wired down only to make possible the initialization. Since the Multics Initializer is now a normal process, these pages can be paged normally.

7. Call Multics $ system_control (BQ.0)

Multics is now in operation and capable of standing alone as an operating system. A return from this call means that the system is to be shut down.

8. Call shut_down

The following actions are taken to shut down the system:

a. Finish any pending I/O

b. Page out the remaining pages

c. Stop all processors