TO: MSPM Distribution
FROM: M. A. Padlipsky
SUBJECT: BL.10.03
DATE: 07/10/68

The attached revision of BL.10.03 supersedes BL.10.03A (02/28/68) as well as the previous version of BL.10.03 (04/24/67). Aside from minor changes, the main points covered are the incorporation of fs_init_3 into the main body of the Initializer, and the documentation of the "uc" data segment and the make_branches procedure. Also, the call to initialize_pwt has been removed, in view of the fact that the pwt has been removed.

Published: 07/10/68
(Supersedes: BL.10.03A, 02/28/68,
BL.10.03, 04/24/67)

Identification

File System Initialization (Part 3)
R. C. Daley

Purpose

This section provides the specification of the procedures which perform the third and final part of file system initialization. These procedures run under the control of the Multics initialization control program during the third part of Multics initialization. The main purpose of this part of file system initialization is to establish branches in the hierarchy for segments already loaded from the system tape and to establish the normal Multics segment fault handling mechanism. After these initialization procedures have been run, the procedures of the Multics initialization control program appear to the file system as an active and loaded process.

Introduction

When the Multics initialization control program first enters the third part of file system initialization, the system is in the following state.

1. All of the segments of the hardcore supervisor have been loaded and all external segment references have been prelinked.

2. The normal file system page fault handler (initialized during part 2 of initialization) is operational and is currently in use.

3. The interim segment fault handler (initialized during part 2 of initialization) is still in use but must be replaced.

File System Initialization

At the appropriate point during the third part of system initialization, the Multics initialization control program takes the following steps to complete the initialization of the file system.

Step 1

File system static storage constants are initialized by means of the following call.

```
     call initialize_fs_static;
```

This procedure initializes the various file system constants in static storage. The device identification, file length and file pointer for the root directory are obtained from the file system device configuration table.

Step 2

The hardcore segment table (HST) is initialized by means of the following call.

```
     call initialize_hst;
```

This procedure creates entries in the HST for all of the segments of the hardcore supervisor as described in the segment loading table. Since the unique identifiers of these segments have not yet been specified, this item is initialized to zero in each HST entry. Since no unique identifiers are yet available, each entry in the hash table is set with the vacant switch ON. The hash table and unique identifiers will be filled in later once branches in the hierarchy are established for each segment in the segment loading table.

Step 3

The known segment table (KST) is initialized by means of a call to the segment control primitive initialize_kst (see BG.3.1). Once this has been done, the KST is modified to reserve all of the segment numbers currently assigned to initializer segments in the segment loading table. This is done to prevent segment control from assigning these segment numbers to other segments to be used during subsequent initialization. These segment numbers are reserved by issuing the following call.

```
     call reserve_init_segs;
```

This procedure expands the length of the KST entry table to allow room to accomodate the highest segment number assigned to any segment listed in the segment loading table. The entries in the entry table corresponding to initializer segments are threaded together in a circular linked list. The pointer to the vacant entry list is then set to indicate that the vacant list is empty. Finally, the highest assigned segment number (highseg) is set to the highest segment number specified in the segment loading table. As a result, segment control will start assigning segment numbers at highseg + 1 and will only assign these reserved segment numbers when specifically requested to do so.

Step 4

An entry is placed in the active segment table (AST) for the root directory segment by means of the following call.

```
     call update_ast;
```

This procedure creates an AST entry for the root directory from the "root_branch" information in file system static storage and updates the AST hash table to point to the new AST entry. The entry-hold count item in this AST entry is set to "1" to prevent the root directory from being deactivated during Multics operation.

Note: During the next four steps, missing-segment faults will occur in manipulating directory segments. These segment faults will be passed to interim_2_segfault which will immediately pass them on to the normal segment fault handler. However, the interim segment fault handler will continue to process segment faults for segments listed in the segment loading table.

Step 5

Note: This step is omitted if the file system hierarchy is known to be intact.

If the file system hierarchy must be restored, the root directory is initialized by means of the following call.

```
     call initialize_root;
```

This procedure simply creates two directory branches in the root directory by means of two successive calls to the directory control primitive appendb. One branch is called "system_root" and defines the subtree to be used exclusively for segments of the Multics initialization control program, hardcore supervisor and the hierarchy reconstruction process (see BH.3.01). The other branch is called "Multics_root" and defines the subtree to be used for all other segments.

<u>Step 6</u>

At this point, the temporary segments used only in the initialization process must be removed, in order that file hierarchy branches not be created for them in the next step. This is accomplished by the following call.

```
     call delete_temp_segs;
```

<u>Step 7</u>

Branches in the file system hierarchy are established for all segments listed in the segment loading table by means of the following call.

```
     call initialize_branches;
```

This procedure basically loops through the SLT, extracting pertinent information for each segment encountered and calling make_branches to create a file system hierarchy branch for each segment. Before the call to make_branches, initialize branches performs a conversion on the "user code" indicated in the SLT, in the following fashion:

```
     usercode = uc$names(n);
```

with declarations

```
     dcl n fixed, usercode char(50), uc$names (16) external
         char(50);
```

where <u>n</u> is the user code from the SLT, and segment <u>uc</u> is an assembled data base which is used to map user codes input as integers from the header file into the 50 character strings required by Directory Control; there is a maximum of 16 possible codes, of which the following two are currently available:

```
     0 = all processes (*.*.*)
     1 = initializer process only (*.system.*)
```

The logic of make_branches is as follows: call appendbx for the segment. If this call is successful the branch has been created and make_branches can continue (see below). If the call is unsuccessful, there are two possible paths: 1) if no access was indicated, this implies that the segment's directory does not yet exist; therefore, call appendbx for as many successively superior directories (indicated in the path name) as necessary. Note that make_branches currently creates directories with ring brackets 32 32 32, as a default case; this is an interim measure, and it is possible that at some future time directories should be ring 0 only. 2) If a branch already existed on the call to appendbx for the segment, change the mode of the branch so that it can be deleted, delete the branch, and then create the branch based on the current information - by another call to appendbx. Regardless of which path was taken after the original call to appendbx for the segment, before returning make_branches must check to see if the branch is multiply-named; if so call chname for the additional name(s).

Upon return from initialize_branches, branches exist in the hierarchy for each segment listed in the segment loading table.

Step 8

The hardcore segment table (HST) is updated to include the unique identifiers of the hardcore segments by means of the following call.

```
call update_hst;
```

This procedure makes successive calls to the directory control primitive status to obtain the unique identifiers for each segment of the hardcore supervisor. As these unique identifiers are added to the appropriate HST entries, the hash table is updated. Upon return from this call the HST is in its final form and ready for normal Multics operation.

Step 9

The AST entries for segments listed in the segment loading are linked to AST entries for their parent directory segments by means of the following call.

```
call link_ast_parents;
```

This procedure takes the following steps for each segment listed in the segment loading table (SLT) which has a corresponding AST entry.

1. A new AST entry for the segment is created by a call to the directory control primitive estblseg, specifying the correct segment number, followed by a call to the segment control utility routine getastentry. There are now two AST entries for the same segment, the new entry just created by getastentry and the old entry pointed to from the SLT.

2. The following items are copied from the new AST entry to the old AST.

   a. aste.id

   b. aste.astparent

   c. aste.xbranch

   d. aste.amtindex

3. The AST hash table is updated to point to the old AST entry and the new AST entry is discarded.

Step 10

At this point, a call is made to seg_fault to allow the interim fault interceptor to be replaced by the normal Multics fault interceptor. When this has been done, the system initializer resumes file system initialization.

From this point on, all segment faults will be processed using the standard segment fault handler.

Step 11

The AST entry-hold counts for segments listed in the SLT with normal status are still set to "1" from part two of file system initialization. Now that the normal segment fault handler is completely operational, these AST entries may be removed from the AST. To reset the entry-hold counts for these AST entries a call is made to the following initialization procedure.

```
call release_ast_entries;
```

This procedure reduces the AST entry-hold count by one for each normal segment listed in the SLT and calls a page control primitive checkentry to determine if the segment should be deactivated at this time.

Step 12

Note: This step is only taken when the file system hierarchy must be reloaded.

If the file system hierarchy must be reloaded, many additional segments must be present in the system hierarchy to accomplish the reload (i.e. segments needed by the hierarchy reconstruction process). These additional segments are loaded from the Multics system tape by means of the following call.

```
call reload_system_hierarchy;
```

This procedure loads each segment found on Multics system tape until the "end-or-reel" indication is reached. For each segment found on the tape, the necessary branches are created in the hierarchy by calls to make_branches, the segment is established in the KST by a call to estblseg, the tape version of the segment is read into the specified segment, the pages are forced out of core by a call to uim$free_core, and the KST entry is removed by a call to makeunknown.

Step 13

Control is returned to the Multics initializer. KST entries currently exist for all segments belonging to the initializer. In effect, the initializer now appears to the file system as an active and loaded process.

Post Initialization Windup

After the remainder of the hardcore supervisor has been initialized and before giving up control to another process, the initializer must again pass control to the file system initializer to return all core which is currently wired down in its behalf. For this purpose the following call is provided.

```
call fs_windup;
```

For each initializer segment listed in the SLT with status other than normal, the corresponding KST entry is located and a call is made to the segment control utility searchast to find the corresponding AST entry. Once the AST entry is found the segment status in the SLT entry is tested and one of the following actions is taken.

1. If the status is wired, a call is made to the page control primitive pcfreecore to release the wired-down pages and the wired-down segment count is reduced by one.

2. If the status is loaded, the page-table-hold count is reduced by one.

3. If the status is active, the AST entry-hold count is reduced by one.

Once the above tests are made and the appropriate actions taken, the AST entry interlock is removed by a call to the page control primitive checkentry. Note that this procedure must respect the interlock strategy since other processes may be running at this time.