Published: 05/24/67

Identification

Segment Control, The Process Load Module
R. C. Daley, D. M. Ritchie

Purpose

The process load module of segment control provides the principal interface between the basic file system and the other modules of the hard-core supervisor. Primitives are provided by this module by which a process may be activated, loaded, unloaded, or deactivated.

Introduction

A process within Multics may be in one of the three following conditions.

a. inactive
b. active
c. active and loaded

When process is inactive, no entries exist for that process in any of the wired down data bases in the hard-core supervisor.

A process is active if there is an entry for that process in the process segment table (PST) and if AST entries exist for the following segments.

a. The process known segment table (KST)
b. The hard-core supervisor stack
c. The process definitions segment (PDF)

An active process may be either loaded or unloaded. A process is loaded if the following conditions are met.

a. The process has a loaded hardcore descriptor segment whose first several pages are wired down. These pages contain the descriptors for all wired-down segments.

b. The entire Process Data Segment (see BJ.1.03) is present in wired-down core.

c. All special segments required by the process are present in wired-down core.

d. The descriptor segment for the last non-hardcore ring in which the process was operating (if any) is loaded and its first page is wired down. This page contains the descriptors for segments which must be accessible in any ring without faults.

To transfer control to a loaded process, it is necessary only to switch to its hardcore descriptor segment. A loaded process becomes unloaded when any of the above conditions are not met.

A process is always activated and unloaded under the control of another process which is currently loaded. However, the loading of a process always occurs under the control of the process being loaded.

Primitives

The following is a list of the primitives provided by the process load module and is followed by a detailed description of each primitive. All of these primitives are privileged to the procedures of the hard-core supervisor.

1. actproc
2. deactproc
3. loadproc
4. loadproc2
5. unloadproc
6. createseg
7. killseg

1. actproc

To activate a process which is currently inactive, the following call is provided.

```
pstep = actproc (dirname,processid);
```

In this call, dirname is the path name of the process directory of the process to be activated and processid is the process identification of that process. Upon return from this call, a pointer to the newly created PST entry for the specified process is returned as the value of pstep.

Upon receiving this call, three successive calls are made to a directory control primitive (estblseg) to find the following segments in the specified process directory and make them known to the current process.

a. the known segment table (KST)
b. the hardcore ring stack
c. the process definitions segment (PDF)

When directory control finds a requested segment, it calls a segment control primitive (makeknown) to make the segment known to the current process. Once the above segments are found and made known in the KST of the current process, a utility routine (getastentry) is called to find (or create if necessary) AST entries for these segments. Once these AST entries are located, the AST-entry-hold counts in the AST entries are incremented by one to insure that the corresponding segments remain active. Then a new PST entry is created for the specified process and linked to the above AST entries and the specified process is now active. However, before returning control to the calling program, the KST entries created during the execution of this call must be deleted by calls to another segment control primitive (makeunknown). Thus, upon normal return from this call, the KST of the current process appears as it did before the call to actproc.

2. deactproc

To deactivate an active process, the following call is provided.

```
call deactproc (pstep);
```

In this call, pstep is a pointer to the PST entry defining the process to be deactivated which cannot be the current process. Upon receiving this call, the AST-entry-hold counts are decremented by one in each AST entry listed in the specified PST entry. If the resulting entry-hold count is zero and the number of pages currently in core for any of these AST entries is zero, a call is made to a page control primitive (removept) to unload the corresponding segment. If the number of pages in core is non-zero, no action is required as the segment will be unloaded automatically when the last page is removed due to inactivity. Before control is returned to the calling program, the PST entry for the now inactive process is deleted.

3. loadproc

To load a process which is currently active but unloaded, the following call is provided.

```
call loadproc (pstep);
```

Again, pstep is a pointer to the PST entry defining the process to be loaded which, in this call, is the current process. Upon receiving this call, the process is using a wired-down descriptor segment prepared by the retiring process. This segment has no DST entry because it was established via a call to createseg. Loadproc creates a DST entry for the descriptor segment, which then is a normal hardcore descriptor segment. Finally a pointer to the new DST entry is placed in the PST entry for the process and the DBR value for the descriptor segment is placed in the interim Process Data Segment.

Next, this routine makes sure that the process KST has been initialized by testing whether its entry count is zero; if the count is zero, the segment control primitive initialize_kst (BG.3.01) is called. If the KST had to be initialized, the process is new (has never been loaded before) so its Process Data Segment must be made known via a call to a directory control primitive (estblseg).

When it is certain that the PDS is known, (whether because it was just established or because it was discovered that the process has been loaded before) the PDS is activated by calling a segment control primitive (getastentry) and read into wired-down core using a page control primitive (pcreadseg). Then the PST entry pointer pstep is placed into the PDS.

Once the PDS is in core, the PDF (Process Definitions Segment; see BJ.1.06) is interrogated to determine whether any special segments are needed by the process; if so, they are established, activated and read into wired-down core, and a collection of pointers to their AST entries is added to the PST entry for the process as a special segment list.

4. loadproc2

If an unloaded process has been loaded before, and if when it became unloaded the first page of a non-hardcore ring descriptor segment was wired down in core, that descriptor segment must be restored before the process can be considered completely loaded. The routine invoked by

```
call loadproc2;
```

checks the Process Data Segment to determine whether a descriptor segment page for a non-hardcore ring should be wired down; if so, loadproc2 uses the Ring Register Simulation Module routine setup_ring to create a descriptor segment for this ring and wire down its first page.

This routine will disappear if ring register hardware is fitted to the 645. See BG.3.05 for a fuller discussion.

5. unloadproc

To completely unload a loaded process, thus rendering the process active but unloaded, the following call is provided.

```
call unloadproc (pstep);
```

Upon receiving this call, the segment-hold count is decremented by one in the AST entry for the PDS of the process to be unloaded. If the resulting value of the segment-hold count is zero, all pages are removed from the wired-down state by a call to core control. If the specified process has any special segments wired into core, they are treated in the same manner as the PDS. Any pages removed from the wired-down state will remain in core until removed from disuse.

All AST process trailers for the specified process are removed from the AST. Finally, any pages or page tables for any descriptor segments, used by the specified process, are returned to core control as free storage and the corresponding DST entries are deleted.

6. createseg

To create a new segment of empty pages (contents all zero) all of which are wired into core, the following call is provided.

```
call createseg (size,max1,priority,descr);
```

In this call, size is the length of the desired segment in words, max1 is the maximum size in words and priority is an integer from 1 to 10 indicating the priority of the request (the lower the value the higher the priority). Upon normal return from this call, a segment descriptor pointing to the newly created segment is returned as the value of descr.

Upon receiving this call, the specified priority is translated into a core "threshold" (see BG.6) to be used in subsequent calls to core control. Segment control then calls core control to assign a page table large enough to accommodate a segment of the specified maximum length. Once the page table is established, core control is called successively to obtain as many pages as necessary to provide the specified segment length. Unused page table entries are then filled with directed faults and control is returned to the calling program.

This call is provided to obtain space for a temporary segment to be used in preparation to loading an unloaded process. A segment created by this call has no corresponding file in secondary storage and thus has no corresponding KST or AST entry.

7. killseg

To destroy a segment which has been created by the preceding call, the following call is provided

```
call killseg (descr);
```

In this call, descr is the segment descriptor of the segment to be destroyed. Upon receiving this call, the page table and pages of the specified segment are returned to core control as free storage.