Identification

Overview of Process Creation, Activation and Loading
R.L. Rappaport

Purpose

This document describes the strategy and structure of the procedures involved in process creation, activation and loading. Collectively these procedures are known as the Process Control Module.

Introduction

A Multics process is characterized by several segments in the File System Hierarchy and by an entry in the Known Process Table. These segments are:

1. A process directory
2. A known segment table
3. A process data segment (which contains a process concealed stack)
4. A process definitions segment
5. A hardcore stack segment

The process directory mentioned above resides in the Process Directory Directory and the other four segments are contained in the process directory itself.

An active process is further characterized by an entry in the Active Process Table, an entry in the Process Segment Table (see section BG.2.00) and by the fact that the process' hardcore stack segment, its process definitions segment and its known segment table are active segments. That is, Active Segment Table entries exist for these segments. The Active Process Table entry points to the Process Segment Table which points to these three Active Segment Table entries.

Finally a loaded process s n ctive process that is further characterized by the fact that its process data segment is currently in core storage and also that the process has a hardcore ring descriptor segment. That is, the process data segment contains information that must be in core storage in order for the process to be considered loaded.

The various parts of the Process Control Module perform the functions that create processes, activate inactive processes, load active processes, unload loaded processes, deactivate active processes and destroy inactive processes.

Process Creation and Destruction

Process creation is accomplished in to phases, one of which is performed by the creating process and the other of which is performed by the created process. The procedures which mape up the two phases are known respectively as create-proc1 and create-proc2.

Basically, create-proc1 creates a process by manufacturing the five basic segments needed by a process and then by making an entry in the Known Process Table for the created process. The created process appears to be a normal, inactive, blocked process and furthermore the call stack of the created process indicates that the apparent call to block originated in create-proc2. Therefore if the new process is ever awakened it will perform a return sequence and find itself executing in create-proc2. Create-proc2 finishes the job of process initialization.

To elaborate more on the above, create-proc1 will first create an empty process directory which is located in the Process Directory Directory in the heirarchy. It will then create branches for the other four segments in this process directory and initialize the segments themselves. The amount of initialization needed for each segment varies. The known segment table and the hardcore stack need no initialization. That is empty versions of these segments are given the new process. The process concealed stack in the process data segment must be filled in with the call history to enable the new process to return to create-proc2. This call history fabrication is accomplished by copying from a template segment created at system initialization (see Section BL.11). The process definitions segment is used by create-proc2 to initialize the new process according to the creator's specifications. This segment, among other things, specifies the linker and search list that will be used by the new process.

Create-proc2 merely initializes the new process according to specifications contained in the process definitions segment. Among other things, create-proc2 makes known, to the new process, the segments that will be used by this process in dynamic linking. Create-proc2 also prelinks these segments for the new process. When complete, create-proc2 calls out to a procedure

named process-init (see Section BO.6.08). However, the search list provided by the creator process, can direct this call to any designed procedure. In this way the creator has complete control over the created process.

Process destruction is basically the inverse of creation. Destruction of a process merely entails destroying the Known Process table entry and appropriate handling of the process' segments. For example, if it is determined that no useful information is contained in the segments they can merely be discarded. Process creation is more fully explained in section BJ.1.01.

Process Activation and De-activation

As was stated above, an active process is characterized by:

1. Active Segment Table entries for the process' hardcore stack segment, known segment table, and process definitions segment. These entries contain flags which insure that the respective segments remain active.

2. A Process Segment Table entry which points to the three entries mentioned above.

3. An Active Process Table entry which, among other things, points to the Process Segment Table entry.

In order to activate an inactive process, some other process must (1) create Active Segment Table entries for these segments (if the entries do not already exist) and place the above-mentioned flags on (2) create a Process Segment Table entry for the process and (3) create an Active Process Table entry for the process. The first two steps are performed by a procedure in Segment Control: actproc (see Section BG.3.03). The arguments used by this procedure are the tree name of the inactive process' Process Directory and the process id of the inactive process. The procedure returns a pointer to the Process Segment Table entry created. At this point an Active Process Table entry is created using the pointer returned by actproc and the Known Process Table entry of the process.

De-activation is simply the inverse of the aforementioned. The Active Process Table entry of the process to be de-activated is destroyed after its Known Process Table entry is updated and the Process Segment Table pointer is obtained. Then entry point deact proc (see BG.3.03) in Segment Control is called passing this pointer as an argument. This procedure destroys the Process Segment Table entry and it also resets the flags that insure that the three basic segments must remain active. These segments will then be de-activated by normal page and segment machinery.

Process Loading and Unloading

The difference between a loaded process and an unloaded but active process is that the loaded process has a hardcore ring descriptor segment and that the loaded process has its process data segment in core storage. The Active Segment Table entry for the process data segment contains a flag which indicates that this segment is wired down. It is in the nature of the active unloaded state that such a process, with a "little bit of help" is capable of loading itself. That is, it can retrieve the process data segment from secondary storage.

Consider an active unloaded process whose execution state is ready (see Section BJ.3.00). This process has no descriptor segment. By virtue of its being in the ready state, the process is liable to be picked for running at any moment. The process that chooses the active unloaded process for running is required to grant the "little bit of help" mentioned above.

Suppose process A is executing in swap-dbr (see Section BJ.5.01) and is trying to switch control to process B. Process B is active and unloaded. Process A will discover that B is unloaded and create both a hardcore ring descriptor segment for B and an interim process data segment for B. Both segments will be created by copying from template segments compiled at system initialization. The creation of these segments for B is the help required. Process A can now transfer control to B.

B will then find itself executing in swap-dbr and it will discover that it is unloaded. It will then call the Process Bootstrap Module (see Section BJ.5.03) using the interim process concealed stack as a stack. This module will retrieve the process data segment from secondary storage and will then return to swap-dbr. At this time B is loaded.

The above example was carried through with an unloaded process that was also in the ready state. Active, blocked processes can be awakened (see Section BJ.3.02) even though they may not be loaded. Awakening is equivalent to changing one's execution status from blocked to ready. Hence, the above example is sufficient.

Unloading of processes is performed in order to free core space. Unloading of a process is accomplished by destroying the hardcore ring descriptor segment of the process and allowing its process data segment to page out automatically. Process activation and loading are more fully explained in Section BJ.1.02.