Published: 05/24/67

Identification

Segment Control, The System Interface Module
R.C. Daley, M.R. Thompson, D.M. Ritchie

Purpose

The system interface module of segment control consists entirely of privileged primitives provided for the use of supervisory procedures and certain administratively privileged processes - e.g., the backup and multilevel system. Primitives are provided by which a segment is made available (or unavailable) to the current process. In addition several primitives are provided for use in servicing segments already known to the current process.

Introduction

Two primitives of the system interface module (makeknown and getdirseg) are provided by which a segment is made known to the current process. If a user wishes to use a segment within his process, one of these two primitives must ultimately be invoked. When this occurs a segment number is assigned to the segment making the segment known to the current process and an entry is made for this segment in the known segment table (KST).

A known segment becomes active only when and if it is referenced by the process, at which time an entry is made for the segment in the active segment table (AST) and the segment is active. Normally, the segment is also loaded at this time (page table assigned) and the referenced page or pages read into core. The remaining pages may be read into core by page control as missing-page faults occur. When, due to relatively low activity, the number of pages in core of a loaded segment drops to zero, page control unloads the segment by removing its page table. Segment control may decide to deactivate an active but unloaded segment by removing its AST entry. AST entries are normally deleted only to make room in the AST for more active segments.

Once a segment becomes known to a process it remains known to that process until an explicit call is made to segment control to make the segment unknown. However, a known segment may enter and leave the active and loaded states many times without any explicit call on the part of the user.

Another primitive of the system interface module (makeunknown) is provided by which a known segment is rendered unknown to the current process. Most of the remaining primitives provide service functions relating to segments already known to the process.

Primitives

The following is a list of the primitives provided by the system interface module and is followed by a detailed discussion of each primitive. All of the primitives are procedures within the hard-core supervisor and may only be called by other hard-core procedures except as indicated.

1. unknown
2. sim2$getdirseg
3. makeunknown (b)
4. segfault
5. boundfault
6. sim2$moveseg (b)
7. sim1$dirmod
8. sim1$branchmod
9. sim1$updateb
10. sim1$unloadseg
11. sim1$deleteseg
12. get_ring (a)
13. sim1$transuse (b)
14. initialize_kst

(a) Callable from the administrative ring.

(b) Callable by backup and multilevel.

1. makeknown

Directory control provides a primitive (estblseg) which locates a branch in the directory hierarchy and makes the associated segment known to the current process. If and when the branch is found, directory control calls segment control to make the segment known to the process by means of the following call.

```
     call makeknown (name,id,mode,ptlist,dirsw,dtbm,dp,
                    slot,dhs,rsw,segptr,slotlist,errcode);
```

In this call, _name_ is the pathname of the branch defining the desired segment, _id_ is the unique identifier from the branch, _mode_ is the effective mode of the segment, _ptlist_ is the segment protection list as specified in the access control list of the current user and determines from which protection rings the segment may be accessed (see section BD.9), _dirsw_ specifies whether the segment is a directory or non-directory segment, _dtbm_ is the date and time indicating when the branch was last modified, _dp_ is a pointer to the base of the directory containing the branch, _slot_ is the index of the branch within the directory, _dhs_ specifies the desired setting of the directory-hold switch in the KST entry to be created, and _rsw_ is the reserved segment number switch (see below). Upon normal return from this call, a pointer (ITS pair) to the base location of the segment is returned as the value of _segptr_. In addition, a list of slot numbers (see BG.7) defining the path of branches from the root directory to the current branch is returned in the array _slotlist_. If, however, the length of the slotlist array is zero, no slot information is returned.

If _rsw_ is ON, then _segptr_ specifies the segment number desired by the caller for the segment being made known; an error return is given if the segment is already known with another number or if the number is unavailable. This feature is used only during system initialization when the Multics trigger is turning itself into a process.

If _rsw_ is OFF, a check is made to determine if any segment already known and listed in the KST has the same unique identifier, using a utility routine (sum$idsrchkst).

If a match is found, and the segment is a directory segment, the new name is appended to the list of names for the entry at which the match was found. If the name is not different from those already present, or if the segment is not a directory, errcode is set to indicate the segment is already known.

If no match is found, the Hardcore Segment Table (HST) (see BG.1) is searched for the unique identifier, and a KST entry is created with the segment number associated with the matching unique ID in the HST.

If the unique ID is not found in the HST, the several unique IDs in the PST entry for the process are searched (see BG.2). These are the identifiers of the KST itself, the process data segment, and the hardcore stack. If a match is found, the associated segment number is used and a KST entry is created.

If both these searches fail, a KST entry is created using the first available segment number. If dirsw is ON, the segment is a directory segment, and the segment name is stored with the KST entry and the transparent-usage switch is turned ON. No name is stored if the segment is not a directory.

2. sim2$getdirseg

In order to search a given directory in the hierarchy, directory control must first obtain a segment number for that directory segment. To establish this segment number from a symbolic directory path name, the following call is provided for the exclusive use of directory control.

```
     call sim2$getdirseg(name, segptr, mode, errcode);
```

In this call name is the symbolic path name of the desired directory segment. Upon normal return from this call, a pointer to the base location of the desired directory segment is returned as the value of segptr and the effective mode of the directory segment is returned as the value of mode.

Upon receiving this call, a utility routine (nsrchkst) is called to search the KST for the specified name. If found, the segment number and mode are immediately returned to the caller. If the name is not found in the KST, a directory control primitive (finddir) is called to attempt to find a branch corresponding to the specified path name. If the branch is found, directory control calls the makeknown primitive to establish an entry for the directory segment in the KST, and returns to getdirseg. Once the KST entry is created, the desired information is returned to the caller.

3. makeunknown

When a process no longer needs a particular segment which is currently known to the process, the segment may be made unknown (no longer known) to the process by means of the following call.

```
call makeunknown (segptr, errcode);
```

In this call, segptr is a pointer within the segment which is to be made unknown. This primitive may be called from procedures in the administrative ring.

Upon receiving this call, the segment number of the segment is extracted from the pointer variable (ITS pair) and used to access the corresponding KST entry to find the unique identifier of this segment. The unique identifier is then used to search the AST to determine if the segment is currently active by calling a utility routine (searchast). If the segment is active, a check is made to determine if the current process is listed with the AST entry as an active user of the segment. If this is the case, the current process is removed from the list of processes actively using the segment, by means of the page control routine setfaults. Since AST entries are normally removed only to make room for new AST entries, no attempt is made to remove the AST entry at this time.

Before returning to the calling program, the KST entry for the specified segment is deleted and directed faults are placed in any segment descriptors belonging to the current process which refer to the segment.

4. segfault

When a process refers to an inactive or unloaded segment, a directed fault occurs in the segment descriptor word of the referenced segment. When this fault occurs, control is immediately passed to a master-mode procedure to process the fault. Normally this procedure calls segment control to process the fault and to activate and load the referenced segment. For this purpose the following call is provided.

```
call segfault (scuptr, dbrptr, errcode);
```

In this call _scuptr_ is a pointer to the location where the processor control unit was stored at the time the fault occurred, and _dbrptr_ is a pointer to the descriptor base register value at the time of the fault. The processor control information includes the segment number of the segment descriptor causing the fault, the address within that segment which the process was attempting to reference, and the ring within which the process was operating when the fault occurred.

Upon receiving this call, a check is made to determine if the fault occurred in the hardcore ring. If so, the HST is examined to determine if it has an entry for the segment number on which the fault occurred. The corresponding unique identifier, whether directly from the HST or in the PST and pointed to by the HST, is used to find the AST entry for the segment, using the utility routine _searchast_. (Thus these segments must always be active.) Using the AST entry, the segment can be loaded and the referenced words read into core. The HST entry for the segment contains the access bits to place in the SDW of the segment.

If the fault did not occur in the hardcore ring or if the referenced segment did not appear in the HST, the KST is examined to find the entry for the faulting segment. If no entry is found, the faulting SDW is checked to determine whether a simulated bound fault (see below) has occurred; if so, _boundfault_ is called. Otherwise, a utility routine (_getastentry_) is called to locate (or create if not found) the AST entry for the desired segment.

Once the desired AST entry is located, a check is made to determine if the effective mode and protection list in the KST entry for the segment is currently up-to-date. This check is made by comparing the date/time-branch-modified item in the KST entry with the date/time-branch-modified in the AST entry. If the time in the AST is more recent than the time in the KST, a directory control primitive (refindb) is called to recompute the effective mode and return the latest effective mode and protection list to segment control. This information is then used to update the KST entry.

A page control primitive (pcreadseg) is then called to insure that a page table is assigned to the segment and the page containing the referenced word is in core. Upon return from this call, the segment is loaded even if it was unloaded before the call. A check is then made to insure that the current process is listed in the AST entry as an active user of the segment by a call to a utility routine (maketrailer).

Once the address of the page table is established, a segment descriptor is created and stored in the descriptor segment of the protection ring in which the process was operating at the time the fault occurred. If the segment is a directory segment, the descriptor access control bits are set to allow only ring zero procedures.

The segment descriptor access control bits for a non-directory segment are determined in the following way.

a. If the ring in which the fault occurred (call it <u>ringno</u>) is higher (less privileged) than the call bracket all access is denied.

b. If <u>ringno</u> is within the call bracket, access is permitted only through procedure calls to prespecified procedure entries by means of the "ring crossing" mechanism (see section BD.9).

c. If <u>ringno</u> is within the access bracket, access is determined according to the effective mode of the segment as specified in the KST entry.

d. If <u>ringno</u> is below the access bracket, access is determined again from the effective mode with the exception that execute permission is denied.

For a more detailed and motivated discussion on segment access and protection see section BD.9 in this manual.

The boundary field of the descriptor word is prepared in cases c and d above. If the append permit is given, the boundary is set to allow the entire segment (up to its maximum length) to be accessed. If the append permit is not given, the boundary is set to allow the segment to be accessed only up to its current length.

The 645 hardware, when checking for a bound fault, tests only whether the page number of the referenced word is greater than the number in the boundary field of the SDW for the segment. Thus zero-length segments cannot be represented in the boundary field of an SDW.

To allow detection of the accessing of a word in the first page of a segment whose length is currently zero, a missing-segment fault is placed in the SDW, and a special code is used in the boundary field, which is normally zero for SDWs of missing segments. When this code is detected by segfault, boundfault is called to simulate the fault.

In preparing a segment descriptor word, segment control may make use of the following directed faults.

1. All access denied - This fault is used to indicate that the process has no access to the segment.

2. Ring-crossing fault - This fault is used to indicate that the segment is accessible only by calling prespecified procedure entries.

3. Incompatible access fault - This fault is used in cases when the effective mode cannot be represented within the framework of the descriptor access bits. This case arises when the read attribute is OFF and the write or append attributes are ON.

4. Simulated bound fault - This fault, which is actually a missing segment fault with a special code in the boundary field, is used to represent zero-length segments.

6. <u>boundfault</u>

The 645 takes an "out-of-bounds" fault on paged segments in two cases.

1. The process has used a segment number so large that it refers to a descriptor segment page whose number is larger than that in the DBR boundary field.

2. The process has used an internal address so large that it refers to a page whose number is larger than that in the boundary field of the SDW for the segment in which the internal address lies.

The first case is always an error, but the second may or may not be a true error. To sort out the possibilities, the following call is available to the interceptor for out-of-bounds faults and to segfault (which catches "simulated bounds faults"; <u>cf</u> discussion of segfault above.)

```
call boundfault (scuptr, dbrptr);
```

Here <u>scuptr</u> is a pointer to the SCU information <u>dbrptr</u> is a pointer to the DBR at the time of the fault

This routine first checks the SCU information to determine if the processor was attempting to fetch an SDW when the fault occurred (that is, case 1 above applies); if so an error is returned.

If the process was operating in the hardcore ring at the time of the fault, then the number of the segment whose attempted access caused the fault is examined to determine whether the segment is a hardcore segment (appears in the HST). If so, an error is returned because the SDWs for such segments always have a boundary field corresponding to the maximum length of the segment, and any reference beyond the maximum length of any segment is always an error.

If both these tests fail the KST entry for the segment is examined; if the process has append permission an error is again returned because as in the last case the boundary field of such segments reflect their maximum length.

If the process does not have append permission the boundary field reflects the actual segment length at the time of creation of the SDW. In this case the AST entry for the segment is obtained using a utility routine (searchast). If the referenced portion of the segment is beyond the current segment length given in the AST entry, an error is returned; but if the referenced page is within the current segment length, the SDW boundary field is updated to reflect the new length and a normal return is given. This last possibility arises because some other process with append permission may have increased the length of the segment subsequent to the creation of its SDW in this process.

The remaining possibility is that the attempt by searchast to find an AST entry for the segment is a failure; this may occur if the segment is deactivated by some other process between the instant the fault occurs and the searching of the AST. By the time the "not found" return from searchast occurs the SDW for the segment will contain a missing segment fault, so boundfault simply gives a normal return. When the immediately subsequent segment fault takes place, the true segment length will be used to create the new SDW. Resumption of the instruction causing the segment fault may or may not cause an out-of-bounds fault but at this point the segment is active so an out-of-bounds fault can be handled.

5. sim2$moveseg

The multilevel system (see section BH.1) operates as a privileged process within Multics. This process decides, on the basis of activity, when a segment should be moved to another on-line secondary storage device. When multilevel decides to move a segment it first makes the segment known to the process in the usual way. Once the segment is known, multilevel causes the segment to be moved by issuing the following call.

```
     call sim2$moveseg(segptr, did, errcode);
```

In this call, segptr is a pointer to the base of the segment to be moved and did is the device identification of the device to which the segment is to be moved.

Upon receiving this call, a utility routine (getastentry) is called to insure that the segment is active. This routine is passed the identification of the specified device which is used to indicate in the AST entry that the segment is being moved. Upon return from this routine a page control utility routine (pcreadseg) is called to insure that at least one page is in core. This call serves to trigger the automatic page-move mechanism. While the page is being read into core, control is returned to the calling program.

7. sim1$dirmod

Directory control reserves the right to determine when a directory has been modified. This strategy allows directory control to record only those modifications to a directory which represent a significant change to the logical structure of the directory. Insignificant modifications, such as the setting and resetting of internal interlocks, need not be recorded. When directory control wishes to indicate that a directory has been modified, it calls the following primitive.

```
     call sim1$dirmod(segptr, errcode);
```

In this call, segptr is a pointer to the directory which has been modified. Upon receiving this call, segment control calls a utility routine (getastentry) to find the AST entry for the specified directory segment. Once the AST entry is obtained, segment control merely calls a page control primitive (updates) to update the date and time last used and last modified items in this AST entry and all its superiors.

8. <u>sim1$branchmod</u>

Whenever directory control makes a change to a branch which might affect the access rights of any user, segment control is informed by means of the following call.

```
     call sim1$branchmod(id, dtbm, errcode);
```

In this call, <u>id</u> is the unique identifier of the branch and <u>dtbm</u> is the date and time when the branch was modified. Upon receiving this call, a utility routine (searchast) is called to determine if the associated segment is currently active. If an entry for this segment is found in the AST, the date-and-time-branch-modified item in the AST entry is replaced by the new <u>dtbm</u>. A page control primitive (setfaults) is then called to place directed faults in all segment descriptors currently pointing to the segment. If and when these faults occur, segfault will recompute the access rights from the latest information in the branch.

9. <u>sim1$updateb</u>

At certain times, directory control may wish to update a branch of an active segment with the more-current information in the segment's AST entry. For this purpose, the following call is provided.

```
     call sim1$updateb(id, segptr, errcode);
```

In this call, <u>id</u> is the unique identifier of a branch defining an active segment and its AST entry, and <u>segptr</u> is a pointer to the directory within which the branch resides. Upon receiving this call, a utility routine (searchast) is called to locate the desired AST entry from the unique identifier. Once the AST entry is located, the desired information is extracted from the AST entry and a directory control primitive (wrbranch) is called to update the branch. If no AST entry is found corresponding to the specified unique identifier, it is assumed that the segment has been deactivated and a normal return is given to the caller.

10. <u>sim1$unloadseg</u>

In order to cause a segment to be unloaded or deactivated, the following call is used.

```
     call sim1$unloadseg(id, deactsw, errcode);
```

In this call, id is again the unique identifier of a segment and its AST entry. If deactsw is ON, the segment will be unloaded, then deactivated; if it is OFF, the segment will be unloaded but not deactivated.

Upon receiving this call, a utility routine (searchast) is called to locate the AST entry from the unique identifier. If no AST entry is found, it is assumed that the segment is already inactive and a normal return is given. If it is found, the segment is unloaded by use of a page control primitive (cleanup) to remove any pages in core and unload the segment when the number of pages drops to zero. If deactsw is OFF, return is made at this point. Otherwise another utility routine (delastentry) is called to deactivate the segment and delete its AST entry.

A segment which is to be deactivated by this routine (deactsw ON) must be a terminal node in the hierarchy--that is, it must either be a non-directory segment, or, if it is a directory segment, there must be no inferior active segments.

11. sim1$deleteseg

When directory control wishes to delete a branch in the hierarchy, the associated segment must first be deleted. To accomplish this, the following call is provided

```
call sim1$deleteseg(segptr, errcode);
```

In this call, segptr is a pointer (ITS pair) to the base of the segment to be deleted which must be a terminal node in the hierarchy.

Upon receiving this call, the segment number is extracted from segptr and used to locate the KST entry of the specified segment. A pointer to this KST entry is then passed to a utility routine (getastentry) to locate (or create) the AST entry of the segment. Once the AST entry is found, a page control primitive (pctruncate) is called to truncate the segment to zero length. Finally, the AST entry is deleted by calling unloadseg, with deactsw ON.

12. get_ring

In order for the gatekeeper (c.f. BD.9.01) to determine the ring to which control should be switched on an inter-ring call, the following call is provided:

```
call get_ring (callptr, oldring, newring);
```

Here _callptr_ and _oldring_ are supplied by the caller and are respectively a pointer to the desired entry point and the number of the ring in which the call took place. The argument _newring_ is returned and is the ring to which control should be switched.

To compute _newring_, get_ring examines the call and access brackets of the segment corresponding to _callptr_ by examining its KST entry (see BG.1). If _oldring_ is below the call and access bracket the lower bound is returned; if _oldring_ is within the call bracket but outside the access bracket then get_ring checks that the offset specified by _callptr_ corresponds to a legal entry point, or gate. If so, the upper bound of the access bracket is returned. If _oldring_ is outside the call bracket or within the access bracket an error return is given.

13. sim1$transuse

Frequently, it is necessay for processes of the backup and multilevel system to use segments without affecting the date-and-time-last-used of these segments. To accomplish this, the following call is available exclusively for the use of processes of the backup and multilevel system (see section BH).

```
     call sim1$transuse(segptr, tus, errcode);
```

In this call, _segptr_ is again a pointer to the base of the segment and _tus_ is the desired setting of the transparent-usage switch to be stored with the KST entry of the segment. Upon return from this call, the previous value of the transparent-usage switch is returned as the value of _tus_.

14. initialize_kst

Before any segments can be made known by entering them in the KST, the KST entry and hash tables must be initialized. The following call is provided for this purpose:

```
     call initialize_kst;
```

This routine merely allocates storage for the minimum size id and name hash tables and sets the "vacant" switch in each entry ON. The minimum size entry table is then allocated, the vacant switches are turned ON, and the entries are threaded together as described in BG.1. This routine is called by a Process Load Module routine (loadproc) when the process is begun. It is intended that the smallest KST entry table be sufficiently large so that it will usually not have to grow, since this is a time-consuming operation.