Published: 05/24/67

Identification

Segment Control, The User Interface Module
R. C. Daley, M. R. Thompson

Purpose

The user interface module of segment control consists entirely of unprivileged primitives designed for both system programs and the general user. These primitives perform service functions relating to segments which are already known to the process.

Introduction

In order to use a primitive of the user interface module, the segment to which the primitive refers must already be known to the process. In all calls to the user interface module the related segment is specified by a pointer (ITS pair) to a location within the segment. The segment number is extracted from the ITS pair and used to locate the KST entry corresponding to the segment. If the specified segment is not currently known to the process, an error is reflected to the calling program by setting the errcode parameter to the error number.

Primitives

All of the primitives of the user interface module are unprivileged and are available to both system programs and the general user. The following is a list of the primitives of the user interface module and is followed by a detailed discussion of each primitive:

1. uim$read_seg
2. write_seg
3. uim$free_core
4. uim$truncate_seg
5. uim$core_test
6. uim$check_access
7. check_ring

1. uim$read_seg

The following call is provided by which a process may declare that a portion of a segment is to be needed shortly.

```
call uim$read_seg(addptr, nwords, errcode);
```

In this call, addptr is a pointer to the first location of a block of core of size nwords. Segment control reserves the right to completely ignore this call. However, if the call is not ignored, a utility routine (getastentry) is called to locate (or create if not found) the AST entry for the specified segment. Once the AST entry is found, a page control primitive (pcreadseg) is called to start reading in any pages which fall within the specified area and which are not already in core.

2. write seg

There is no descriptor setting which allows writing (appending) but not reading. The following call is provided to accomplish the writing (appending) for a process which has write (append) permit but does not have read permit.

```
call write_seg(addptr,array,nwords,errcode);
```

In this call addptr is the pointer to the address of the first word to which the write is directed, and nwords is the count of the number of words to be written from array.

The effective mode is checked to determine if the write_seg is valid. If it is valid, the specified number of words from array is copied into the segment.

This call is intended for the use of the file system interface module (see section BF.4).

3. uim$free core

The following call is provided by which a process may declare that part of a segment is no longer needed at this time.

```
call uim$free_core(addptr,nwords,errcode);
```

Again, addptr specifies the beginning of an area of core of size nwords. Segment control reserves the right to completely ignore this call. However, if the call is not ignored, a utility routine (searchast) is called to search the AST to determine if the segment is active. If no AST entry is found, or if the segment is currently

active but unloaded, no further action is taken and control is returned to the calling program. If the segment is loaded, and the current process is the only process listed as an active user of the segment, a call is made to a page control primitive (pcfreecore) to begin removing any pages which reside entirely within the specified area.

4. uim$truncate_seg

To truncate a segment to a shorter length by discarding information at the end, the following call is provided.

```
call uim$truncate_seg(addptr,errcode);
```

In this call, addptr is a pointer to the first word to be discarded. Upon receiving this call, two utility routines (getastentry and getloaded) are called in succession to insure that the segment is active and loaded. A call is then made to a page control primitive to truncate the segment. Use of this call requires the write permission in the ring of the calling program.

5. uim$core_test

To determine if a specified area within a segment is currently in core, the following call is provided.

```
percent = uim$core_test(addptr,nwords,errcode);
```

Again, addptr specifies the first location of an area of nwords in length. Upon return from this call, the percentage of the specified area which is currently in core is returned as the value of percent. The percentage is returned as an integer from zero to one hundred.

Upon receiving this call, a utility routine (searchast) is called to search the AST for an entry for the specified segment. If no entry is found or if the segment is unloaded, the percentage returned is zero. If the segment is currently loaded, the percentage is found by calling a page control primitive pctestcore.

6. uim$check_access

To obtain the effective access rights to a specified segment with respect to a given protection ring, the following call is provided

```
mode = uim$check_access(addptr, ringno,errcode);
```

In this call, segptr specifies a segment and ringno specifies the ring for which the access rights are computed. Upon return from this call, the access rights are returned as an effective mode (REWA) which is computed in the following way.

1. If ringno is higher (less privileged) than the segment access bracket, all attributes are OFF.

2. If ringno is within or below the access bracket, the attributes are set with the effective mode. (It is assumed here that the caller is interested in determining the effective access rights and is not concerned about whether or not an attempt to execute a segment results in a ring crossing.)

7. check ring

Whenever arguments are passed by inter-ring calls, care must be taken to be sure that any access the called procedure makes to the arguments is permitted in the ring from which the call took place. The most important illustration of this necessity occurs when an outer-ring procedure calls an inner ring procedure which writes into its argument. Then if the argument, accidentally or otherwise, happens to point to some important system data base which is writable in the inner ring, and if no precautions are taken, the inner ring procedure will change the data base in some unexpected way; the results are likely to be catastrophic.

In order to check whether a given set of segments is accessible from a specified ring, the following call is available.

```
call check_ring(segptrarray, ringno, errcode);
```

Here segptrarray is an array of pointers (ITS pairs) and ringno is the number of a ring. The procedure check_ring determines whether each segment specified by segptrarray can be accessed from ring number ringno (that is, whether the upper bound of the access bracket of each such segment equals or exceeds ringno). If so, then errcode is set to zero. If not, or if some of the segments do not exist in the process, then errcode is set to non-zero.