Chapter Six
Phase Three: The Support Level

1. Introduction

For the next level of your operating system, you are to design and implement support for virtual memory and other facilities that can be used by a simple timesharing system. The timesharing system will support up to five terminals. Associated with each terminal is a single process, called a T-process, that will execute programs submitted through that terminal. In addition, several system processes will be required in order to handle paging, the pseudo-clock and disk I/O.

We want a way for each T-process to request system services without changing the nucleus or compromising system security. This suggests implementing new SYS operations. These are outlined in the next section.

However, several problems need to be addressed when providing this support:

(a) I/O — a process that can issue I/O operations could specify any memory location for data transfer and can thus overwrite any location it wishes.
Solution: make sure the page frame containing the device registers is not accessible to the T-processes.

(b) Nucleus implemented SYS's — by specifying random locations as, say, semaphores, a process could alter any location it liked.
Solution: run the T-processes in user mode, since they are then prohibited from issuing nucleus implemented SYS's. (This is why the nucleus specification contained this restriction.)

(c) Arbitrary memory accesses using stores, moves, etc.
Solution: use memory mapping hardware to give each process a private address space disjoint from both the support level and other T-processes. (Actually, the specifications do provide for a segment to be shared by all T-processes, and a malicious process could interfere with others that were trying to, say, synchronize. However, this is not a security hole, merely a nuisance.)

Each T-process will run in user state, with interrupts unmasked, and will have an address space consisting of two segments: segments 1 and 2. (Recall that the EMACSIM memory management supports thirty two segments numbered 0 - 31.) Segment 2 is shared by all of the T-processes, segment 1 is unique for each. Thus T-processes communicate and synchronize by using segment 2.

T-processes cannot be assumed to be trustworthy. Thus, it will be prudent for you to construct this level so that it protects itself from T-processes. For one thing, this will make it easier to debug; T-processes will be stopped before they completely destroy the integrity of the system.

The system processes of this level will run with memory mapping on, in kernel mode and with interrupts unmasked. The code and data for these processes will reside in segment 0 together with the nucleus. These processes will always be resident in memory; the pages of segment 0 will not be paged out. A missing segment or missing page memory management trap for an address in segment 0 while a system process is running will denote an error condition.

Since this is a level-structured system, it should not be necessary for you to make major changes to your nucleus in order to complete this new level. The details of how to load the nucleus and support level are covered in the next chapter.

Recall that your nucleus started running a process named *p1* after it had completed initialization. Thus, you should name the code that is to receive control to initialize this level *p1*.

2. New Instructions

The following ‘‘operations’’ are supported by this level. T-processes request these operations by calling the appropriate SYS functions. Recall that executing a SYS causes control to be transferred to the nucleus SYS handler. But, if the appropriate SYS5 instruction has been executed for the process that caused the SYS trap, then a higher level trap handler can receive control. Thus, among the first things a T-process should do after it has been created are three SYS5 instructions (one for each type of trap it can generate). As the T-processes themselves are not trustworthy, when the support level is creating a T-process, it should start a process that does the appropriate SYS5’s and some other initialization (more on this in the next chapter) and then does a LDST to transfer control to the ‘‘real’’ T-process.

The pseudo-instructions supported by this level are:

Read_from_Terminal (SYS9):

Requests that the invoking T-process be suspended until a line of input has been read from the associated terminal. The data read should be placed in the virtual memory of the T-process executing the SYS9, at the (virtual) address given in D3 at the time of the call. The count of the number of characters actually read from the terminal should be available in D2 when the T-process is continued. An attempt to read when there is no more data available should return the negative of the ‘‘End of Input’’ status code in D2. If the operation ends with a status other than ‘‘Successful Completion’’ or ‘‘End of Input’’ (as described above), the negative of the Status register value should be returned in D2. Any negative numbers returned in D2 are ‘‘error flags.’’

Write_to_Terminal (SYS10):

Requests that the T-process be suspended until a line (string of characters) of output has been written on the terminal associated with the process. The virtual address of the first character of the line to be written will be in D3 at the time of the call. The count of the number of characters to be written will be in D4. The count of the number of characters actually written should be placed in D2 upon completion of the SYS10. As in SYS9, a non-successful completion status will cause an error flag to be returned instead of the character count.

V_Virtual_Semaphore (SYS11):

Performs a V operation on a semaphore whose virtual address is in D4 at the time of the call.

P_Virtual_Semaphore (SYS12):

Performs a P operation on a semaphore whose virtual address is in D4 at the time of the call. If a T-process passes virtual addresses in D4 for SYS11 and SYS12 that do not refer to segment 2 it should be terminated.

Delay (SYS13):

On entry, D4 contains the number of microseconds for which the invoker is to be delayed. The caller is to be delayed at least this long, and not substantially longer. Since the nucleus controls low-level scheduling decisions, all that you can ensure is that the invoker is not dispatchable until the time has elapsed, but becomes dispatchable shortly thereafter.

Disk_Put (SYS14) and **Disk_Get** (SYS15):

These instructions are provided to perform *synchronous* I/O on the DISK0 device. Some form of seek reduction algorithm should be implemented in the disk driver process in order to facilitate high disk utilization. On entry, D3 should contain a virtual address from (to) which the disk transfer is to be made. D4 should contain the disk track number and D2 the sector within the track to (from) which the data transfer will be made. The corresponding write (read) operation of the 512-byte data is initiated, and only after it has completed is the T-process resumed. The number of bytes actually read or written replaces the contents of D2. As for the SYS9 and SYS10 instructions, the negative of the Status register is returned in D2 instead of the byte count if the operation does not complete successfully. Note that the disk controller of the EMACSIM assumes that the memory buffer involved in the I/O is a *physical* address.

Get_Time_of_Day (SYS16):

Returns the value of the time-of-day clock in D2.

Terminate (SYS17):

> Terminates the T-process. When all T-processes have terminated, your operating system should shut down. Thus, somehow the ‘‘system’’ processes created in the support level must be terminated after all five T-processes have terminated. Since there should then be no dispatchable or blocked processes, the nucleus will halt.

3. Design

The support level is divided in two parts. In the first part you will implement the memory mapping system and in the second part you will implement the virtual memory system. You will service only a subset of the new system calls. There will only be two test processes that will exercise your page fault, delay and termination handlers. A detailed discussion follows:

When a page fault occurs, you need to find a free page frame in memory, and if necessary page in. It will be easier to debug your system if there is no paging, and this is achieved by allocating more page frames than needed by the testing processes.

When a memory management fault occurs for the first time for a page, it is inefficient to page in a blank page from the swap device. It is better to find a free page frame and simply mark the page as present. The reference bit can be used to determine if a page has ever been used. During initialization all reference bits can be set to 1, then if a page fault occurs and the reference bit is 1, there will be no need to page in.

A process and virtual memory should be created for terminals 0 and 1. Segment 1 of each T-process can have up to 1024 pages, but we will use only the first 32 pages (page 0 - page 31). Page 31 is used as the stack page and is also used to load the bootcode. For each T-process, the bootcode should be copied to a free page frame. This should be done during the initialization phase and page 31 should be marked as present and modified.

In the first part you will write only the functions that service SYS9, SYS10, SYS13, SYS16 and SYS17. You should provide empty functions for the other support level system calls.

In your cs580/support directory create two directories called part1 and part2. In the part1 directory create two files: support.c and slsyscall1.c. In the part2 directory create two files: page.c and slsyscall2.c . A general discussion of each file follows:

support.c:

This module coordinates the initalization of the support level, it services traps that are passed up, and it creates any necessary system processes. It has the following functions: p1(), p1a(), slsyshandler(), slmmhandler(), slproghandler(), tprocess() and cron().

void

p1():	This function initializes all segment and page tables. In particular it protects the nucleus from the support level by marking the nucleus pages as not present. It initializes the page module by calling pageinit(). It calls getfreeframe() and loads the bootcode on the page frame. It passes control to p1a() which runs with memory mapping on and interrupts enabled.
void static	
p1a():	This function creates the terminal processes. Then it becomes the cron process.
void static	
tprocess():	This function does the appropriate SYS5s and loads a state with user mode and PC=0x80000+31*PAGESIZE.
void static	
slmmhandler():	This function checks the validity of a page fault. It calls getfreeframe() to allocate a free page frame. If necessary it calls pagein() and then and it updates the page tables. It uses the semaphore sem_mm to control access to the critical section that updates the shared data structures. The pages in segment two are a special case in this function.
void static	
slsyshandler():	This function has a switch statement and it calls the functions in slsyscall1.c and slsyscall2.c
void static	
slproghandler():	This function calls terminate().
void static	
cron():	This function releases processes which delayed themselves, and it shuts down if there are no T-processes running. cron should be in an infinite loop and should block on the pseudoclock if there is no work to be done. If possible you should synchronize delay and cron, otherwise one point will be lost.

slsyscall1.c

This module handles some of the support level system calls. It has the following functions: readfromterminal(), writetoterminal(), delay(), gettimeofday() and terminate().

page.c

This module controls the page frames in memory. It has four functions: pageinit(), getfreeframe(), putframe() and pagein(). This module will be provided.

void static
pageinit(): This function initializes variables to point to the free page frames.

int
getfreeframe(): This function blocks on the semaphore sem_pf if there are no free page frames. This semaphore contains the number of free page frames. The arguments are the T-process number, the segment and the page, it returns a free page frame.

void
putframe(): This function releases the page frames of a process that died. It is called by terminate().

void
pagein(): This function is called by slmmhandler() if paging is required. The arguments are the T-process number, the page , the segment and the page frame. This function cannot hold sem_mm while doing IO.

slsyscall2.c This module handles the rest of the support level system calls. All of these functions are empty: virtualv(), virtualp(), diskput() and diskget().

4. The Bootcode

You should load the following code at the beginning of page 31.

```
int bootcode[ ] = {
        0x41f90008,
        0x00002608,
        0x4e454a82,
        0x6c000008,
        0x4ef90008,
        0x0000d1c2,
        0x787fb882,
        0x6d000008,
        0x10bc000a,
        0x52486000,
        0xffde4e71
};
```

This array declaration is available as the file ˜cs580000/share/hoca/support/bootcode.c for you to copy. These hex numbers happen to correspond to the object code of the following ‘‘bootstrapping loader’’ written in assembly language (this is presented here only for your information):

```
41f9 0008 0000      lea       0x80000.l,a0
2608                mov.l     a0,d3
4e45                trap      #0x5
4a82                tst.l     d2
6c00 0008           bge.w     0x8
4ef9 0008 0000      jmp       0x80000.l
d1c2                adda.l    d2,a0
787f                moveq.l   #127,d4
b882                cmp.l     d4,d2
6d00 0008           blt.w     0x8
10bc 000a           mov.b     #10,(a0)
5248                addq.w    #1,a0
6000 ffde           bra.w     -0x22
4e71                nop
```

It causes a program to be read in from the terminal associated with the current T-process, and then executed by jumping to 0x80000, the beginning of each T-processes text. To start the execution of the bootcode you should set the PC to 0x80000 + 31 * PAGESIZE.

5. Testing the Support Level

There are two test programs for testing the support/part1 level. These programs are available in the ˜cs580000/share/hoca/support/part1 directory under the names *termin0* and *termin1*; You should link to these files in your support/part1 directory. Don’t copy them because we might find the need to change these files. The C code for the test programs are also available in the same directory under the names *t0.c* and *t1.c* for you to look at. The two programs are:

1) termin0 — contains general tests for various SYS's, paging.
3) termin1 — tests the delay and paging.

6. Handling Traps in T-Processes

The support level should take some sane action if a T-process causes a trap that it was not expecting. A non-exhaustive list of such traps includes:

- Memory management traps for pages other than those in segments 1 or 2.
- Program traps.
- Attempts to execute undefined SYS's.
- Passing unreasonable parameters to SYS instructions implemented in this level.

The response to an unreasonable action on the part of a T-process should probably be to terminate it — treat the action as equivalent to a SYS17. Note that T-processes should be run in user state with memory mapping on, and therefore should not be able to address directly either the nucleus or the support level address space. Therefore, they cannot subvert the functions implemented by the nucleus or this level by directly modifying instructions or data structures associated with them. Nor can they execute privileged instructions such as LDST, HALT, etc. Also note that the handlers for the T-processes and the handlers for the system processes might be different due to the distinct nature of these processes. More on this in the next chapter.

Chapter Seven
Notes on the Support Level

1. Compiling and Loading

The files of the nucleus and the support level should be compiled separately using the **-c** option and then loaded as follows:

egcc.sol2 -i *nucleus .o files crt1.o support level .o files libutil.a* **-o** hoca

This will produce an executable file called *hoca* with the code and data loaded as

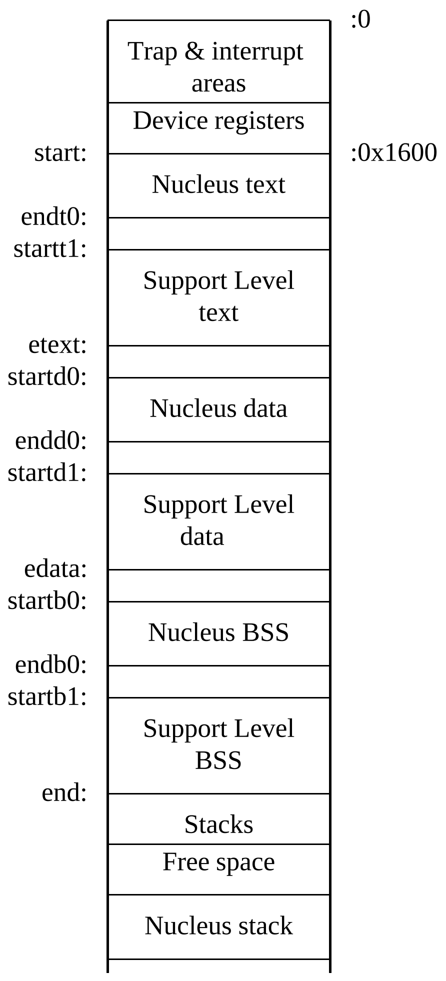


Figure 1. *EMACSIM Memory Map*

shown in Fig. 1.

The C-compiler generates three areas as a result of a program compilation:

(1) The *text* area. This is where the instructions of the program are put in.

(2) The *data* area. This is where the program variables with an *initialized* value are placed.

(3) The *bss* area. Contains the storage for the program variables with no explicit initial value (they get value 0 as a default).

Storage for these three areas are allocated in the order given above.

The six areas (two of each of the three types) will be loaded in memory so that there is one full page of separation between them (these areas will *not* necessarily begin on page boundaries, but are guaranteed to begin on a *different* page than the end of the previous are). This is so that you can protect the nucleus from the support level by making the areas associated with it unavailable when memory mapping is turned on (as it will be case when the support level runs). There are external labels as shown in Fig. 1 associated with the beginning and end of each of the six areas. You can refer to them symbolically in your program after having declared them as external *functions*. Refer to §6 in Chapter 5 where we discussed how to initialize a variable with the address of a function.

2. Using Memory Mapping

The memory mapping hardware (explained in the *Documentation for the* EMACSIM *Computer System*) can be used effectively by the support level to help protect the nucleus from trusted process errors.

Consider the addresses used by the support level if memory mapping is off (that is, in the code produced by the **egcc.sol2** compiler). These addresses should be real addresses, since the support level may need to pass them to the nucleus (semaphore addresses, for example). As shown above, the nucleus and the support level will occupy the pages of memory after the device registers. Viewing support level addresses as virtual addresses, they would be in segment 0. Thus, if a process runs correctly under such conditions, it will still run correctly with memory mapping *on* if it uses a page table that simply maps virtual address *v* to physical address *v*. (All system processes can of course use the same segment and page tables.)

An advantage of this scheme is that the page table used by system processes and support level handlers can have the pages containing the nucleus marked as ‘‘not present.’’ Since a properly functioning process should never attempt to access a location inside the nucleus, this should cause no problems. On the other hand, if a runaway process should attempt to access the nucleus, it will generate a memory management trap and fail.

Admittedly, this scheme does nothing to prevent a malicious system process from damaging the system; however, our concern here is with accidents, not sabotage.

As the T-processes don't have segment 0 they can not address the nucleus or the support level (provided the page frames assigned to the T-processes come from the area marked as ''free space'' in Fig. 1).

3. T-process and Protection

Consider what happens inside the nucleus if a process running in user mode generates a trap.

Suppose a process running in user mode does indeed generate a SYS trap. The state of the process is saved in the SYS old area, the new state is loaded, and the nucleus begins running. The nucleus finds that the process that issued the SYS was in user mode, so the SYS is treated as a software trap and, if the process has done the appropriate SYS5, is passed up. The new state will be loaded from the ''new'' area of SYS trap state vector as specified by the SYS5. We would like to have some sort of prespecified routine (the handler) to handle the trap and, for example, perform I/O to the terminal. In a similar fashion, if the process generates a legal ''missing page'' memory management trap, the handler should arrange for the page to be brought into memory.

The problem here, of course, is if the process is running in user mode, how could it have done the SYS5's in the first place? For that matter, how could we be sure that the ''new'' area specified is trustworthy? The answer to this lies in process creation. If a process does a SYS1 to create another process whose SR specifies user mode, then that process could never do a SYS5. However, a process running in kernel mode *can* do a LDST to go into user mode. Suppose that as part of system initialization we run a trusted kernel mode process for each terminal. This process does the SYS5's (and any other necessary housekeeping tasks) and *then* does a LDST to transfer control to the untrustworthy user program (and go into user mode). Then, a trap generated by the user code will cause control to be transferred to the location specified by the SYS5, which is presumably trustworthy. The handler will take any necessary action, then return control to the user code by doing a LDST from the SYS5 old area.

It is important to realize that there is *only one* process associated with each terminal. The nucleus remains blissfully unaware that all of these LDSTs are being done, and knows only about the original SYS1. Thus, these ''T-processes'' lead a rather schizophrenic existence; part of the time they are executing user-supplied code in user mode, while the rest of the time they are executing the trusted, system-supplied trap handler.

4. Practical Considerations

Suppose a T-process wants to read a line from its terminal. It does a SYS9, specifying a (legal) address in segment 1. The handler does the I/O, and must then move the line of input into the user's memory space. It could examine the page table to determine where this is, but a simpler solution is to give the interrupt handler access to the page tables used by the user code and do a straightforward copy to move the data to the desired location. The segment tables of the support level handlers and of the T-processes are thus the same except that the handler has access to segment 0 while the T-processes do not.

One minor subtlety here is that the handler's attempt to move the data to the user space could cause a page fault, but as long as the page fault handler gets no traps, and it shouldn't, this recursion can only be one level deep. The implication is that the support level handler for SYS and program traps will have to save the corresponding SYS5 old area somewhere else.

5. Understanding Memory in the Support Level

5.1. The Problem

When running the support level, one should realize that we are now dealing with three sets of data:

1) nucleus data
2) support level data
3) user data.

All of these data have to be located somewhere in memory and should somehow be kept secure and separate from other data. Furthermore, with regard to support level data, one should bear in mind that there will be several T-processes running, and one must ensure that each is capable of maintaining its support level data secure from those of the other T-processes. For instance, there may be two T-processes requesting to read from their respective terminals and one function which performs the terminal reads. We want both T-processes to be able to call this function concurrently without there being interference between the two with regard to the local data of that function. To see how this can be accomplished, let's look at how the C-compiler sets up our programs upon compilation and where the data is located.

5.2. Object Code

As indicated in the Fig. 1, the C-compiler generates two segments† of output: text (code) and data (for the purposes of this chapter we consider the ''data'' and ''bss'' areas together). The text segment contains all the executable instructions that constitute the program that was compiled, and only those instructions. The data segment contains all data that are declared externally to any function. These data are static and are retained between function calls. Data that are local to specific functions do not appear in either of

† Here we are using the word ''segment'' in a generic sense meaning a group of related locations. These ''segments'' should not be confused with the segments of the memory mapping hardware of the EMACSIM.

these segments. Space for these data is allocated dynamically *on the stack* when the corresponding function is called.

5.3. Reentrant Code

A function is considered *reentrant* if more than one process can be executing it concurrently. We wish to make the functions that make up the support level handlers reentrant so that any of the T-processes may be executing them at the same time. To do so, the data that are private to each T-process must be kept disjoint.

The data that are local to one such function will be placed on the stack when that function is called. As long as we ensure that each T-process keeps its support level stack disjoint from the others, we can be sure that there be no interference. We can enforce that each T-process has its support level stack disjoint from those of the others by the way we set up the states which are specified as trap vectors; more on that below.

There may be data that should be private to a given T-process but that are not local to one specific function. For instance, we will probably want each T-process to have one specific semaphore on which it will block when it must stop processing, e.g., when it is waiting for delay, I/O, etc. This semaphore must be accessed by several different functions, including those not in the support level handlers (for example, the clock process must be able to V this semaphore to release processes which delayed themselves). This semaphore cannot be local to any function and instead of being on the T-process's private stack it will be in the data segment as set up by the compiler. Because there will be more than one T-process running, we will have to explicitly declare more than one such semaphore. In general, we will want to declare an array of variables (indexed by T-process) for every variable we wish to be private to a T-process that is *not* local to a specific function.

5.4. Implementing Reentrant Code—Keeping Stacks Disjoint

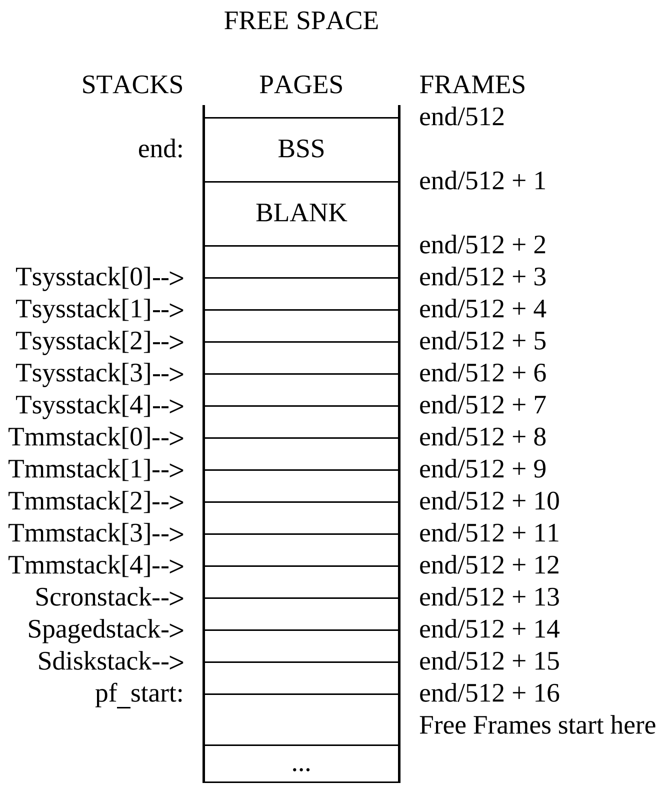
As noted earlier, the stacks for the support level handlers will have to be disjoint between the T-processes. Recall that the different interrupt and trap handlers in the nucleus could all share the same stack (i.e., the new states loaded upon a trap or interrupt all had the same SP). This was for two reasons. First, no more than one process could be executing in the nucleus at a given time, and when that process exited the nucleus it was finished with whatever it had to do (in the nucleus) at that time. Secondly, it was not possible for a process in the nucleus to generate a trap (actually program traps were possible, but these indicated a serious problem with the nucleus itself), and all interrupts were masked. Therefore, a process could not ‘‘reenter’’ the nucleus while it was already in it.

The situation with the support level is quite different. For one thing, there are several T-processes running at one time, and no assurance that when one process enters the support level, none of the others will also enter it concurrently. Therefore, each T-process needs its own support level stack. Also, a T-process executing in the support level could cause another support level trap to occur. The only way this can happen is if a process executing in the support level SYS handler causes a memory management trap (consider the case where a Tprocess tries to copy data from a terminal read to a segment1 virtual address that is not in physical memory). Therefore the stack used by the memory management trap handler in the support level must be disjoint from that of the SYS (and program) trap handlers.

Stack corruption occurs when a stack overflows its preallocated space and corrupts data structures, variables or other stacks. Stack corruption is one of the most difficult problems to diagnose, the value of global or local variables may suddenly change.

To prevent stack corruption you should use memory mapping to avoid stack overflows. Each stack page should be on a page boundary and should be marked as "present". Each stack should be allocated 512 bytes. The pages above and below the stack page should be marked as "not present". That way if there is a stack overflow, a memory trap will occur and stack corruption will be prevented.

Each T-process should have two separate stack pages. Each system process should have its own stack page. There should be 8 different segment0 tables and also 8 different page tables for segment0. There should be 13 different stack pages after the BSS as shown in the diagram below:



The "end" label can be anywhere in a page. However "end/512" will be on a page boundary. The stacks like Tsysstack[0] are two bytes less than the page boundary and should grow towards low memory.

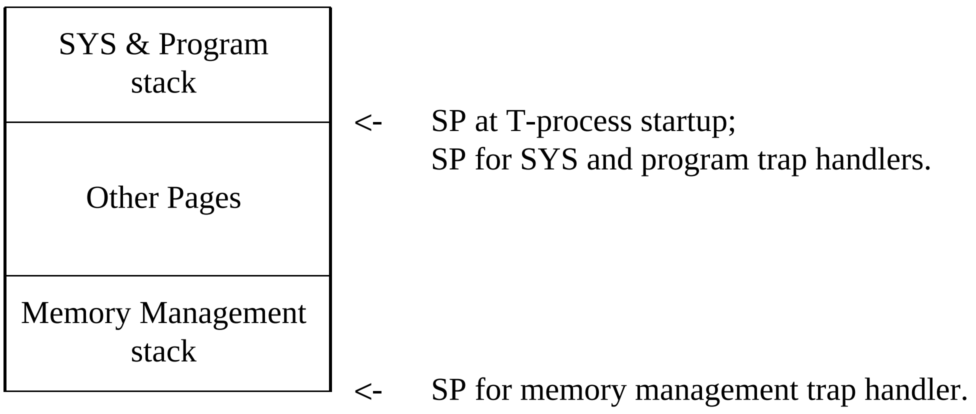
The function pageinit() will be provided to you. This function is in the file page.c in the hoca/support/part2 directory. pageinit() will initialize Tsysstack[], Tmmstack[], Scronstack, Spagedstack and Sdiskstack. This function will also initialize frame arrays and variables for your convenience.

The size of each page table for segment 0 should be 256, for segment 1 the size should be 32 and for segment 2 the size should be 32.

Our root process (‘‘p1a’’ — the process that will actually create the T-processes) will execute a loop that will, with each iteration, spawn a new T-process. Each T-process will be in supervisor mode as it starts up because it has to do its SYS5’s, etc. In setting up the states for the created T-processes, the root process should give each a disjoint stack. This can be done by setting the CRP register and also the SP register in the state of the created process:

```
<create a generic start state for T-processes>;
for (each terminal) {
      < set the CRP to the Segment Table for this terminal in start state>;
      < set SP to Tsysstack[terminal] in start state>;
      <create process>;
}
```

It is then the responsibility of the initialization code for the T-processes to specify separate stacks, one for the SYS and program trap handlers, and one for the memory management trap handler. A T-process’s stack area (as allocated by the root process) will end up looking like this:


SYS & Program stack
Other Pages
Memory Management stack
<- SP at T-process startup; SP for SYS and program trap handlers.
<- SP for memory management trap handler.

These tables show the areas that should be marked as "Present" in the page tables for Segment0 for each process:

PROCESS	EVT	SEG0 PAGE2	TRAPINT AREAS	DEVICE REGS	NUCLEUS TEXT	SEGMENT0 SUPPORT TEXT	NUCLEUS DATA	SUPPORT DATA	NUCLEUS BSS	SUPPORT BSS
p1a/cron	No	Yes	No	No	No	Yes	No	Yes	No	Yes
Tprocess priv-mode	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes

PROCESS	Tsysstack[i]	Tmmstack[i]	p1/p1a STACK	NUCLEUS STACK
p1a/cron	No	No	Yes	No
Tprocess[i] priv-mode	Yes	Yes	No	No

Each Tprocess in supervisor mode should only map its two stack pages and it cannot map the stack pages of another Tprocess. Tprocesses in user mode do not have access to Segment0. You need to mark Page2 in Segment0 as "Present" because this page is used by memory mapped registers.

5.5. Implementing Reentrant Code—Non-local Data

As outlined above, a good solution of maintaining data private to a T-process that is not local to a specific function is to make an array indexed by the number of T-processes. If this is done, however, there is the problem of how to select the correct element of this array. That is, each T-process must somehow know its terminal number when it begins executing. One solution is to keep the terminal number in a register; we suggest D4. When creating the T-processes, the root process can ensure that each T-process starts up with its terminal number in D4. The function that begins the T-processes can initialize its trap vectors (set up with SYS5) by putting its terminal number into D4 in the new states that will be loaded up on a support level trap. In this way, any of the support level trap handlers will ‘‘come alive’’ with their terminal number in D4. After this, the terminal can either be kept in D4 (if you’re careful - remember you sometimes need D4 to pass information to the nucleus) or passed as a parameter to other functions that may be part of the support level handlers.

5.6. Shared Data

There will, of course, be data that must be shared between the different T-processes. For instance, the ‘‘Delay’’ SYS call probably places the invoking T-process on a queue which the clock process will periodically examine. We need to be able to maintain the integrity of data structures such as this queue. To do so we can use a very simple solution: use semaphores implemented by the nucleus to guard entry to critical sections. With every shared data structure we will associate a semaphore which will control access to any operations on the data structure.

5.7. User Data in the Test Programs

Each of the test programs has its own data, some of which is private to the T-process. However, there is no reason to worry about these private data of the T-processes. They will be in segment one (private), and the virtual memory system will keep them from conflicting with those of the other T-processes.

5.8. Miscellaneous

Make sure that the system processes have a virtual map that is identical to the physical map.

Remember that the SYS1 - SYS8 functions push the system call number on the stack and execute a trap 3. SYS9 - SYS17 don’t work in the same way because the SP contains a virtual address and exception processing executes with memory mapping turned off. Each new SYS call corresponds to a trap.

```
SYS9  - trap  5
SYS10 - trap  6
SYS11 - trap  7
SYS12 - trap  8
SYS13 - trap  9
SYS14 - trap 10
SYS15 - trap 11
SYS16 - trap 12
SYS17 - trap 13
```

Remember to load the addresses of STLDSYS9 - STLDSYS17 in the EVT.

5.9. Custom Test Programs

It is your responsibility to devise appropriate tests for your code, and understand that it will be graded with a variety of test conditions beyond what is listed in the documentation.

Your Support phase should be well tested. You can test your Support phase with different timeslices and conditions. You can also write and compile your own custom test programs, please see:

/home/cs580000/share/hoca/support/compile-termin

However you are responsible for debugging your custom programs. Please make sure that your custom program does not use too much memory.