Chapter Five
Phase Two: The Nucleus

1. The Nucleus

The nucleus described below is similar to the T.H.E. system constructed by Dijkstra, and to the one discussed in lecture. The purpose of the nucleus is to provide an environment in which asynchronous sequential *processes* exist, as well as to provide synchronization primitives for these processes. This implies that the nucleus must do low-level CPU scheduling. In addition, the nucleus provides facilities for ''passing up'' conditions such as program traps, memory management traps and certain system call traps to the next level in the level structure.

The nucleus phase is divided in two parts. In the first part the nucleus handles traps and in the second it handles interrupts. The abstract machine implemented by the nucleus is described in the following sections. The instructions implemented by the nucleus are invoked when a process calls the corresponding SYS function. In addition, traps cause the nucleus to take actions. In particular, the nucleus transforms SYS, program, and memory management traps into procedure calls.

The nucleus has frequent occasion to deal with regions of memory containing a 76-byte copy of the processor state, i.e., D0-D7, A0-A6, SP, PC, SR, and the CRP registers. This will be referred to as a *processor state*.

2. First Part: Traps

The nucleus adds processes to the ready queue and allows the process at the head of the queue to execute. In the first part a process will execute until it finishes or until it blocks itself on a semaphore. This scheme is not desirable and it will be improved in the second part by using clock interrupts.

In writing your nucleus, you may assume that no more than 20 (this constant should be equivalent to the mnemonic ''MAXPROC'') processes will exist at any time. All addresses the nucleus deals with may be assumed to be physical (as opposed to virtual). The nucleus should preserve the state of a process; if a process has memory mapping enabled when it is interrupted or calls a nucleus-implemented (SYS) function, then it should still be in that state when it continues executing.

3. Handling Traps

After the nucleus has initialized itself when the machine is first turned on, the only mechanism for entering it is through a trap. As long as there are processes to run, the machine is executing instructions on their behalf and temporarily enters the nucleus long enough to handle traps when they occur.

Traps generated by certain SYS functions cause the nucleus to perform some service on behalf of the process executing the instruction. The specification of these services is detailed in the next section. SYS traps other than the ones listed below, as well as all program traps and memory management traps may be ‘‘passed up’’ by the nucleus to the process responsible for them. From the point of view of a process, this mechanism simulates a software interrupt by saving the old processor state in memory and loading a new state from a different place in memory. These two memory addresses (one for the old state and another for the new state) together constitute the process's *trap state vector*. This is initialized by a process when it executes a SYS5 instruction. The level above the nucleus will provide the handlers for these types of traps.

4. New Instructions

The following functions (SYS1 - SYS8) are handled directly by the nucleus. They are executable only by processes running in supervisor state. The nucleus should confirm this and in case of execution by a user state process, the request should be treated as a privileged instruction program trap without executing the requested operation. That is, the appropriate SYS request will not be executed and the process's *program trap state vector* will be used as outlined below.

Create_Process (SYS1)

When executed, this instruction causes a new process, said to be a *progeny* of the first, to be created. D4 will contain the address of a processor state area at the time this instruction is executed. This processor state should be used as the initial state for the newly created process. The process executing the SYS1 instruction continues to exist and to execute. If the new process cannot be created due to lack of resources (for example no more entries in the process table), an error code of -1 is returned in D2. Otherwise, D2 contains zero upon return.

Terminate_Process (SYS2)

This instruction causes the executing process to cease to exist. In addition, any progeny of this process are terminated (and, by implication, the progeny of the progeny, etc.). Execution of this instruction does not complete until *all* progeny are terminated.

Sem_OP (SYS3)

When this instruction is executed, it is interpreted by the nucleus as a set of V and P operations atomically applied to a group of semaphores. Each semaphore and corresponding operation is described in a *vpop* structure. The *vpop* structure is defined in "vpop.h". D4 contains the address of the *vpop* vector, and D3 contains the number

of *vpop*s in the vector.

NOTUSED (SYS4)

If this instruction is executed, it should should result in an error condition. Your nucleus should halt if this instruction is executed.

Specify_Trap_State_Vector (SYS5)

When this instruction is executed, it supplies three pieces of information to the nucleus:

- The type of trap for which a *trap state vector* is being established. This information will be placed in D2 at the time of the call, using the following encoding:

  0 - program trap
  1 - memory management trap
  2 - SYS trap

- The area into which the processor state (the old state) is to be stored when a trap occurs while running this process. The address of this area will be in D3.
- The processor state area that is to be taken as the new processor state if a trap occurs while running this process. The address of this area will be in D4.

The nucleus, on execution of this instruction, should save the contents of D3 and D4 (in the process table entry) to pass up the appropriate trap when (and if) it occurs while running this process. When that happens, the nucleus stores the processor state at the time of the interrupt in the area pointed to by the address in D3, and sets the new processor state from the area pointed to by the address given in D4.

Each process may execute a SYS5 instruction *at most* once for each of the three trap types. An attempt to execute a SYS5 instruction more than once per trap type by any process should be construed as an error and treated like a SYS2.

If a trap occurs while running a process which has not executed a SYS5 instruction for that trap type, then the nucleus should treat the trap like a SYS2 (Terminate_Process).

Note that only instructions SYS1 to SYS8 are treated as *instructions* by the nucleus. The other SYS instructions (SYS9 to SYS17) are passed up as SYS traps.

Get_CPU_Time (SYS6)

When this instruction is executed, it causes the CPU time (in microseconds) used by the process executing the instruction to be placed in D2. This means that the nucleus must record (in the process table entry) the amount of processor time used by each process.

Wait_for_Clock (SYS7)

This instruction performs a P operation on the pseudo-clock semaphore. This semaphore is V'ed every 100 milliseconds by the nucleus.

Wait_for_IO_Device (SYS8)

This instruction performs a P operation on the semaphore that the nucleus maintains for the I/O device whose number is in D4. All possible devices for the EMACSIM are assigned unique numbers in the range 0-14 as shown in the *const.h* file of Appendix 2. The appropriate device semaphore is V'ed every time an interrupt is generated by the I/O device.

Once the process resumes after the occurrence of the anticipated interrupt for printer and terminal devices, D2 should contain the contents of the Length register for the appropriate device and, for all devices, D3 should contain the contents of the device's Status register. These two registers constitute the I/O operation completion status (see above), and may have already been stored by nucleus. This will occur in cases where an I/O interrupt occurs before the corresponding SYS8 instruction.

5. Design

Create a directory called nucleus under your cs580 directory. Under nucleus create two directories: traps and interrupts. In traps create three files: main.c, trap.c and syscall.c. Copy the file ~cs580000/share/hoca/nucleus/interrupts/int.c to your interrupts directory. A general discussion of each file follows, note that RQ means Ready Queue.

main.c:

This module coordinates the initialization of the nucleus and it starts the execution of the first process, p1(). It also provides a scheduling function. The module contains two functions: main() and init(). init() is static. It also contains a function that it exports: schedule().

void

main(): This function calls init(), sets up the processor state for p1(), adds p1() to the RQ and calls schedule().

void static

init(): This function determines how much physical memory there is in the system. It then calls initProc(), initSemd(), trapinit() and intinit().

void

schedule(): If the RQ is not empty this function calls intschedule() and loads the state of the process at the head of the RQ. If the RQ is empty it calls intdeadlock().

trap.c:

This module handles the traps, it has the following static functions: trapinit(), trapsyshandler(), trapmmhandler() and trapproghandler().

void trapinit():	This function loads several entries in the EVT and it sets the new areas for the traps.
void static trapsyshandler():	This function handles 9 different traps. It has a switch statement and each case calls a function. Two of the functions, waitforpclock and waitforio, are in int.c. The other seven are in syscall.c. Note that trapsyshandler passes a pointer to the old trap area to the functions in syscall.c and int.c
void static trapmmhandler():	
trapproghandler():	These functions pass up the traps or terminate the process.

syscall.c:

This module handles system calls. There are seven functions corresponding to seven system calls: creatproc, killproc, semop, notused, trapstate, getcputime, and trapsysdefault. The system calls are described above.

int.c:

The functions in this module will be updated in the second part. This module will be provided to you, it contains the following functions: intinit(), waitforpclock(), waitforio(), intdeadlock() and intschedule().

6. Initial Startup

It is *your* responsibility to test your nucleus. We will supply a C function called *p1()* (source is in ~cs580000/share/hoca/nucleus/traps/p1.c for you to look at that exercises some of the primitives to help us evaluate your nucleus. You can compile each file of your nucleus with the **-c** flag to **m68k-elf-gcc**. This will produce a ".o" file corresponding to the ".c" file.

The assembly routines that we provide to facilitate your work are in the directory ~cs580000/share/hoca/util. The assembly source is in the *.s files and all the object files (*.o) have been archived in the lib/libutil.a library.

The functions *verhogen* and *passeren* are a front end for the system call *semop*. These functions are also included in lib/libutil.a. They take a semaphore pointer as an argument. The source code is in the ~cs580000/share/hoca/util directory.

To load all these files along with the test program, you should use the Makefile that runs the m68k-elf-ld linker.

Then you can run the file *nucleus* on EMACSIM. Remember to declare the test function as ''external'' in your program by saying

```
extern int p1();
```

before you refer to it. You can initialize a variable to contain the starting address of a function by simply naming the function in the right hand side of an assignment statement. For example, to initialize a processor state area so that the function *p1()* is invoked when it is loaded as the new state, you can say

```
state_t new;
    ...
new.s_pc = (int)p1;
```

where *s_pc* is the program counter field of the processor state structure.

There is a new file in the ~cs580000/share/hoca/h directory. The util.h file has a list of the functions in the libutil.a archive. The functions must be declared as extern int if your nucleus is going to use them. Your nucleus source should have the following line:
#include "../h/util.h"

In the initialization stage, your nucleus should load the addresses of the functions in libutil.a in the correct location of the *EVT*. For example, the address of the *STLDZERO* function should be loaded at memory location 0x014. This can be done in the following way:

```
*(int *) 0x014 = (int) STLDZERO;
```

When a division by zero occurs, *EMACSIM* looks at the contents of memory locations 0x014-0x017. This value is loaded in the *PC* and thus the routine *STLDZERO* is executed. *STLDZERO* stores the processor state, i.e. all the registers of the machine in an old trap area. It also stores the exception vector number in the old area. Then *STLDZERO* loads the processor state from the new area. Your nucleus should have initialized the new area to point to a trap handler which you should have written.

The numbers below each register in Figure 1 tell you the offset of each register from the beginning of the area. Each old and new area is 76 bytes long. All registers are 32 bits long except the SR, thus two bytes are unused. These two bytes form an area called

d0	...	d7	a0	...	sp	pc	sr/tmp	crp
0	...	28	32	...	60	64	68/70	72

Fig. 1 *Format of the Old and New Areas*

tmp and are used by *STLDZERO* to store the exception vector number, in this case the exception vector number is 5. The exception vector number is needed by your trap handler to identify the trap since several program traps will use the same old and new areas.

Please note that storing the processor state has nothing to do with the hardware, that assembly routines that you could have written are doing it, and that it is being done only for convenience.

After loading the *EVT*, your nucleus figures out how much physical memory there is on the machine by examining the SP register. The *new* areas for the traps and interrupts should be initialized so that the nucleus stack starts to grow from the very bottom of physical memory. We suggest that the nucleus be allocated two pages of stack space, but that this size be easily modifiable if needed. After it has completed initializing the necessary data structures, the nucleus should create a single process running the code of the function *p1()* in a state identical to the nucleus with all interrupts disabled. The SP of the new state for this process should be immediately above the nucleus stack. This test process will exercise your nucleus and generate some output in the okbuf buffer to indicate its progress.

7. Accessing Machine Registers Through C

Note that the ‘‘arguments’’ to the various nucleus-implemented instructions are passed through data registers (in particular, D2, D3 and D4). Consequently, to invoke these nucleus services a process must be able to initialize the registers with the actual values of the arguments before issuing the appropriate SYS instruction. This can be accomplished easily through C by declaring the following global register variables:

```
register int d2 asm("%d2");
register int d3 asm("%d3");
register int d4 asm("%d4");
p1()
{
        d4 = devno;
        DO_WAIT_IO();           /* defined to be SYS8() */
        if (d3 == NORMAL) {
                num = d2;
                ...
        } else {
                ...
```

By convention, the C-compiler allocates the first three local register variables to the machine registers D2, D3 and D4, in that order. Therefore, in the code fragment above, we have assigned the variable names d2, d3 and d4 to the machine registers D2, D3 and D4, respectively. Any manipulation of these variables is equivalent to the manipulation of the corresponding register. It is important that this declaration be the first global register declaration and the order of variable names be as given.

When the nucleus finally assumes control after a SYS instruction, the machine registers contain values that were initialized from the SYS trap new area. The register values prior to the SYS trap are stored in the old area associated with this trap. Therefore, the code within the nucleus implementing a certain service can obtain the values of the arguments being passed to it by simply looking at the locations corresponding to the appropriate registers in the SYS trap old area.

8. Termination

The nucleus should detect certain very simple deadlock states. If all of the existing processes are blocked due to P operations on semaphores not associated with I/O devices or the pseudo-clock, the system should shut down with an appropriate message. The nucleus should also shut down (normally) when the last process in the system terminates.