9. Second Part: Interrupts

The nucleus in the first part does not guarantee finite progress. A process executing an infinite loop does not allow the execution of another process. Also it is not a fair scheme, a short process behind a long one would have to wait a long time before it can execute. This problem is solved using the *Interval Timer*. The IT causes an interrupt after a given number of microseconds, it is similar to an alarm. Interrupts work the same way as traps, the *EVT* is checked and a handler executes. This handler can remove a process from the head of the RQ every time that the IT interrupts, and it can add that process to the tail of the RQ. Consequently, every ready process will have an opportunity to execute. You are encouraged to implement CPU multiplexing using a simple round-robin scheduler with time slice value not greater than 5 milliseconds.

I/O devices also cause interrupts, they signal the completion of an operation by interrupting the CPU.

10. Handling Interrupts

After the nucleus has initialized itself, the only mechanism for entering it is through an interrupt or 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 interrupts and traps when they occur.

I/O interrupts should function as V (UNLOCK) operations on the semaphore associated with the appropriate device. You should create an array of semaphores and you should index the array by using the device numbers listed in const.h. Furthermore, the contents of the device's Status register and, if appropriate, Length register, should be saved; this is the I/O operation's *completion status*. It is only necessary to save the most recent completion status for each device. If a process is waiting for I/O for the interrupting device (see SYS8 above), the completion status is passed directly to that process and need not be saved.

The Interval Timer of EMACSIM will be used both for CPU scheduling (enforcing the time slice) and implementing the *pseudo-clock*. This is nothing more than a special semaphore that is V'ed by the nucleus every 100 milliseconds. This mechanism will be used later to implement a general delay facility for processes.

11. Design

In your nucleus/traps/main.c file you should run p1 with interrupts disabled. In your nucleus/interrupts directory you should have 3 files: int.c, intp1.c and Makefile. We will provide intp1.c, note that it is similar to p1.c, but intp1.c tests the wait_for_clock and wait_for_io system calls, and it also tests the interrupt handlers. Some of the processes in intp1.c are different from the ones in p1.c. The Makefile should link some of the .o files in ../traps with int.o, intp1.o and create the object file nucleus. A general description of int.c follows:

int.c:

This module handles interrupts caused by the I/O devices and the clock, and it services two system calls. It contains the following functions:

void
intinit(): This function loads several entries in the EVT, it sets the new areas for the interrupts, and it defines the locations of the device registers.

void
waitforpclock(): This function does an intsemop(LOCK) on a global variable called pseudoclock.

void
waitforio(): This function first checks if the interrupt already occurred. If it has it decrements the semaphore and passes the completion status to the process. Otherwise this function does an intsemop(LOCK) on the semaphore corresponding to each device.

void
intdeadlock(): This function is called when the RQ is empty. If there are processes blocked on the pseudoclock, it calls intschedule() and it goes to sleep. If there are processes blocked on the I/O semaphores it goes to sleep. If there are no processes left it shuts down normally and it prints a normal termination message. Otherwise it prints a deadlock message.

void
intschedule(): This functions simply loads the timeslice into the Interval Timer.

void static
intterminalhandler():
intprinterhandler():
intdiskhandler():
intfloppyhandler(): These functions pass the device type and the device number to inthandler().

void static
inthandler(): This function saves the completion status if a wait_for_io call has not been received, or it does an intsemop(UNLOCK) on the

semaphore corresponding to a device.

void static

intclockhandler():

If the RQ is not empty, this function removes the process at the head of the queue and it then adds it to the tail of the queue. This function does an intsemop(UNLOCK) on the pseudoclock semaphore if necessary and then it calls schedule() to begin the execution of the process at the head of the RQ.

void static

intsemop(): This function is similar to the semop call in the first part. It has two arguments, the address of a semaphore (instead of a state_t), and the operation. This function should use the ASL and should call insertBlocked and removeBlocked.

void static

sleep(): This function is called when the RQ is empty. This function could enable interrupts and enter an infinite loop, or it could execute the "stop" assembly instruction. From C call the asm("stop #0x2000") instruction which loads 0x2000 into the status register, i.e. it enables interrupts and sets supervisor mode, and then the CPU halts until an interrupt occurs. The simulator runs much faster if stop is used.

Software development depends critically on careful testing. 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. For example your solution will be tested with different timeslices and also different conditions.