Objective

In this problem you will learn how to handle I/O service request traps and I/O service completion interrupts. You will see the effects of discontinuities (asynchronous events) on scheduling.

Description

In the prior simulation each iteration of the main loop represented a single quantum of time (time slice). In this simulation each iteration represents a single instruction executing. With each iteration you will be incrementing the PC of the running process by one. The timer interrupt will occur on regular intervals (say every 300 instructions). The PCB will need to be augmented in order to simulate the I/O service traps.

PCB

Figure 1 shows the new fields that need to be added to the PCB class for this simulation.

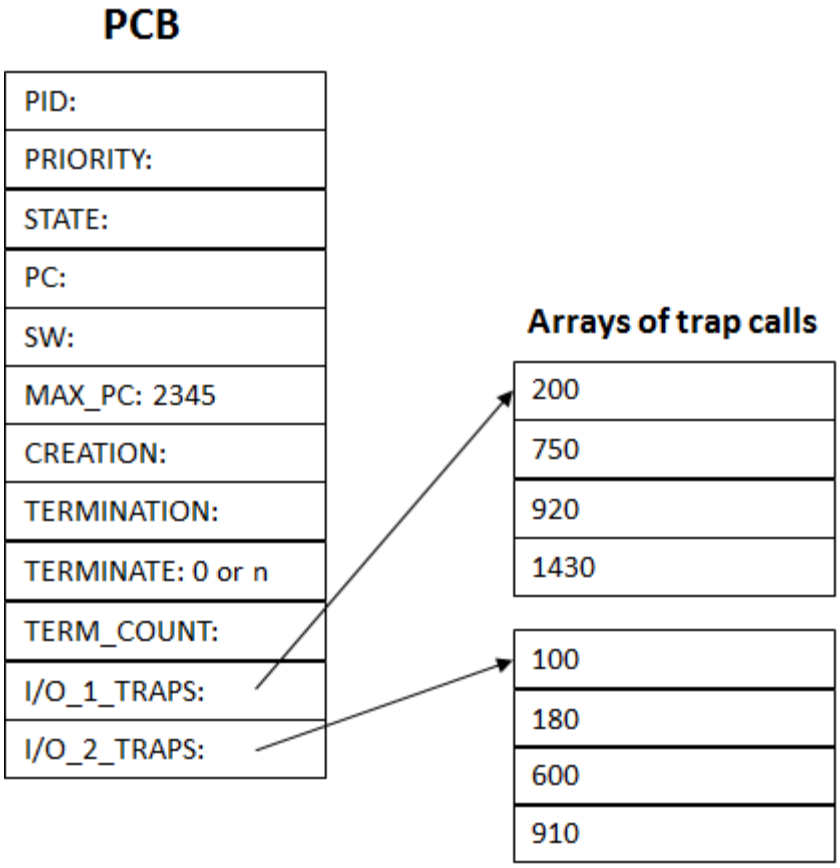

Figure 1. *The augmented PCB*

MAX_PC: This is the integer that gives the number of instructions that should be processed before resetting the PC to 0. In this example the number is set to 2345, meaning that after the PC reaches 2345 it is reset to zero.

CREATION: Computer clock time when process was created. You will need to capture the time when the process is instantiated and store it there.

TERMINATION: Computer clock time when the process terminates and goes into the termination list. If the process does not terminate during the run no time need be recorded.

TERMINATE: This is a control field. If the number is 0 then the process is not one that terminates over the course of the simulation run. If it is greater than 0 then then this is the number of times that the PC will pass its MAX_PC value. For example if this number is set to 15 then the process will terminate after 15 times it passes 2345 execution cycles.

TERM_COUNT: This is the counter to keep track of how many times the process has passed its MAX_PC value.

I/O_1/2_TRAPS: These two fields can be simple integer arrays (inside the struct). Each one contains four numbers representing the PC count where the process will execute and I/O service trap. Each number in each array should be assigned randomly but you must make sure there are no duplicates. The algorithm for invoking the trap call is given below.

The Simulation Architecture

As before the CPU is a loop that represents an execution cycle. However, this time each iteration represents a single instruction execution rather than a chunk of instructions. The running process PC will be incremented by one each time through the loop.

Figure 2 shows a block diagram of the new simulation.

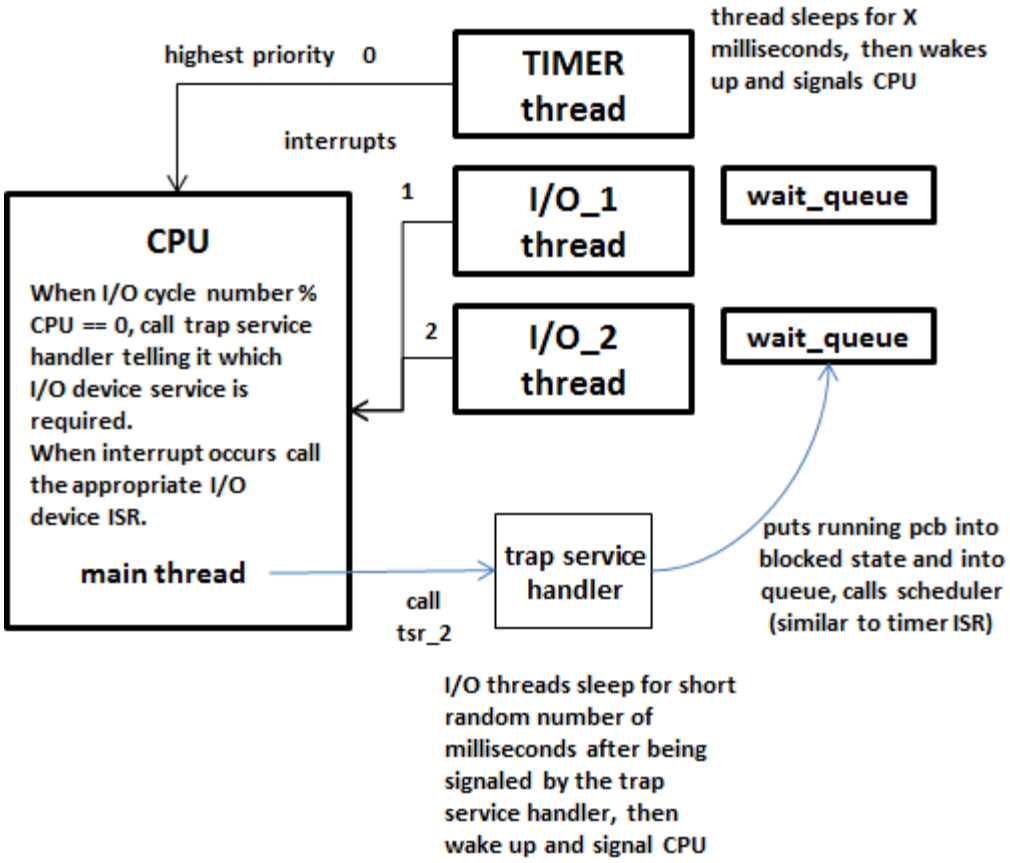

Fig. 2. *The simulation architecture consists of the main thread (see text for explanation) running the CPU and three separate threads representing a timer and two I/O devices. The I/O trap handler works similarly to the timer ISR handler from the last simulation.*

CPU:
As described above this is essentially the same kind of infinite loop from the last simulation with the difference that each iteration represents a single instruction. The CPU thus increments the running process' PC by one each time.

In addition the CPU must compare the PC value with each of the values in the PCB I/O arrays. On a loop where one of those numbers is the same the CPU calls the I/O trap handler passing the trap service routine number (which I/O device is needed). That handler will work very similarly to the timer ISR in terms of taking the running process out of that state and putting it into the waiting queue for the appropriate device. This act also activates an internal timer in the device described below.

When an I/O or timer interrupt occurs the CPU calls the appropriate ISR. That, in turn calls the scheduler just as before. See the scheduler's additional duties below.

If your team is going to try to use pthreads or Java Thread class then note: the CPU is the thread you will start after getting the timer and I/O device threads going (see descriptions below) in the main program.

TIMER:
In this simulation the timer is a separate device that has its own down counter. It can be set to a value that is the quantum (number of execution cycles). Each time the timer is called it decrements. When it reaches zero the call to the function returns a 1 indicating that it threw an interrupt. The CPU then calls the pseudo_ISR for the timer as before.

If your team is going to use pthreads or Java Thread class then note: The timer is an independent thread that puts itself to sleep for some number of milliseconds (the standard sleep function in Linux is in seconds so use the nanosleep() function (time.h) - you may need to experiment with how many the timer should sleep to approximate a single quantum). When it wakes up it will need to "signal" the CPU thread that an interrupt has occurred through the use of a mutex. In the CPU loop use the non-blocking mutex_trylock() call so that the loop doesn't block itself waiting for the timer signal. After throwing the interrupt signal it puts itself to sleep again for the designated quantum. The timer has the highest priority with respect to interrupt processing. It must be accommodated before any I/O interrupt. If an I/O interrupt is processing when a timer interrupt occurs you should call the timer pseudo_ISR from inside the I/O pseudo_ISR to simulate these priority relations.

I/O_x_:
The two I/O devices work exactly the same as the timer but use a random number that should be greater than the quantum time. If your quantum is equal to 300 loops, then your I/O random time number should be 3 - 5 times that. This is so that several processes will get a chance to run while the waiting process is blocked.

However, the I/O device needs to be activated by the I/O service request trap handler from the CPU call. Upon activation the device will down count its random number to zero, whereupon it returns a 1 indicating that it threw an interrupt.

If your team is going to use pthreads or Java Thread class then note: Each time the I/O device is activated by the I/O service request trap handler it goes to sleep for a random number of milliseconds (as above) and when it awakes it signals an interrupt to the CPU thread, in the same manner as the timer signal. Note that the processing of the I/O ISR could get interrupted by the timer so you need to make signalling provisions for this.

Program Structure

Figure 3 shows the non-threaded CPU loop. Note the order of the calls to various timer, I/O devices. These are set in this order to observe priorities of interrupts.

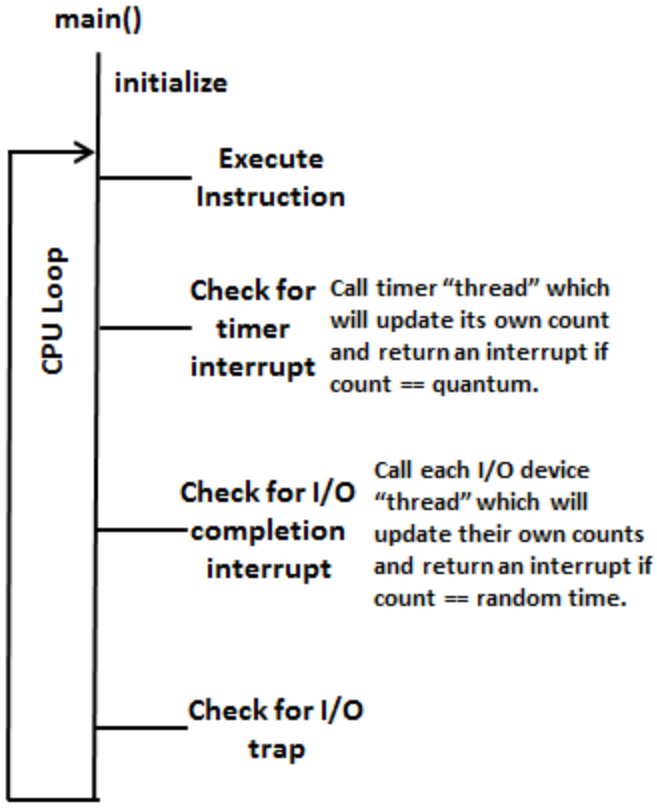

Fig. 3. *The main cpu loop (thread). Each of the "Check" for interrupts can either be a call to the devices or can refer to receiving a signal from the device threads.*

The program starts by initializing objects and some starting PCBs in the new_queue. Upon entering the loop it "executes" an instruction. This just means that the CPU increments the PC of the running process. The three "check" steps will work differently depending on whether or not you are writing the program with pthreads or Java Thread class. If not, then the check for timer interrupt is actually a call to a function that downcounts its quantum variable (decrement the counter). If the variable is not zero the function returns a zero. If the counter reaches zero, the function resets the counter to the quantum value and returns a 1 signifying that an interrupt has been thrown.

If you are writing a threaded program the timer is an independent thread as described above. The check timer interrupt in the CPU loop examines a pthread-protected mutex to see if an interrupt flag has been sent by the timer (see description of mutex uses below). If it has then the CPU loop calls the pseudo-ISR for the timer which works just like the last simulation.

After completing the check for the timer interrupt the loop checks to see if any I/O device has sent an interrupt because it completed its task and the PCB in the front of the waiting queue needs to go back into the ready queue. In the non-threaded version this is a call to the I/O device (1 & 2) which is also downcounting an integer to zero. This integer is initialized by an I/O service trap (below). When it reaches zero, as with the timer, it returns a 1, otherwise it returns 0.

The threaded version is essentially just like the timer described above.

The loop then checks to see if an I/O service request is pending on this execution (corresponding with a TRAP instruction). That means it examines the trap calls arrays of the current running process to see if the current PC value is one of the instruction counts in the arrays. If so then the CPU calls the I/O trap handler which takes the running process out of that state and puts it into the waiting queue for the device (array one has PC counts for I/O device one). This is essentially the same as for the timer. The simulated device sets a downcounter to a random value that is several times larger than the system quantum. Each time the device is checked for an interrupt it downcounts this variable as described above.

Threaded Programs

If you are using pthreads or the Java Thread class, your timer, and the two I/O devices are each separate threads that act independently. Each is an endless loop (use for(;;) { syntax for efficiency). When the thread is started it should initialize its own variables (i.e counters if used and waiting queue on the I/O devices). It then enters the endless loop and does the work described above.

There are two ways you can implement these threads. The most time efficient way is to have the threads sleep (see the Gnu page on sleeping and look at the nanosleep function call: http://www.gnu.org/software/libc/manual/html_node/Sleeping.html

On some systems this could actually lock the whole process but if it only locks the thread you should use it. Check your system's documentation to be sure or try writing a small two threaded program in which one loop prints a system time message and the other thread executes sleep - see if there are gaps in the time printed out.

The other way is to make the threads do the same thing as the simulated devices. Each time they loop they will decrement a counter till it reaches 0. They then signal the CPU thread using mutexs (note that there is a non-blocking mutex lock call: pthread_mutex_trylock(). Use this in the CPU thread so that it does not block when checking its interrupt signal. See: https://computing.llnl.gov/tutorials/pthreads/

Output

You will be providing the same sort of output as in the last assignment except that this time you will also print out the trap and I/O interrupt events as they occur. All printing should be done in the CPU loop.

Turn-in:

As before, zip your files up and one person only submit. Please follow the guidelines I posted re: the last assignment and what should be in the files.