TCSS 422, Fall, 2017

Final Project

PURPOSE:

In this project you will complete the simulator by adding a few more things to the code you have already built and doing some experiments to see differences in behaviors under different conditions. Additionally, in this project you will experience using the pthread library to produce more efficient simulation code.

Producing and Running Simulations

This exercise will be a mild form of an incredibly important activity in industry, the *simulation of machines and software* that might otherwise be extremely costly to build and test. Simulations tell us a lot about the behavior and dynamics of complex systems and companies at the forefront of their product markets are turning to simulation increasingly. When you complete this assignment you will have gotten a reasonably good taste for doing simulation studies, something you can put into your resume. But you will also have a much better understanding of how a complex OS works and much more practice writing useful code.

REQUIREMENTS:

There are a number of additional requirements above those from the problems. In this simulation we will be adding synchronization services such as mutexes and condition variables. Figure 1 is an updated version of the figure from Problem 4 with the addition of synchronization services (e.g. mutexes). Only one such service mechanism is shown. You will be implementing as many as are needed based on the mix of process types (see below).

Priority Scheduling

Use the MLFQ scheduling algorithm from Problem 4.

Process Creation

Below are the descriptions of “realistic” processes that will be “running” in the simulation. These will be created similarly as in Problem 4. Generate new processes only periodically (say every time ***S*** is activated). Now, however, you will be generating a mix of these realistic processes. In the case of communicating processes these will need to be generated in pairs. Use a random distribution of the four types. Note that communicating pairs cannot be terminatable.

CPU, TIMER, I/O DEVICES AS PTHREADS

The main loop is simulating the CPU and is itself a thread. When the program first starts up you will create the other three threads (see Figure 1). The timer thread should not start doing anything until the CPU loop is started. Design it so that the CPU can signal it to start. This can be done after the first few processes are created and the scheduler runs for the first time.

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.

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 signaling provisions for this.

See the paragraphs below for more information on implementation of threads for this project.

REALISTIC PROCESSES:

The processes you created thus far were just to populate the OS with dummy tasks in order to exercise the Multi-Level Feedback Queue mechanism. In this project you will actually craft a few "working" processes to add to the mix. This means these new process types will simulate doing realistic work such as requesting I/O and service requests similar to Problem 4. But this time they will be requesting services for the purpose of synchronization and communications.

I/O Processes

You already have these processes. In the final problem simulation though you should limit the number of these in the mix to no more than 50 total I/O processes - with dynamic creation and termination as before. This is not different from Problem 4, but these processes will not be the only types.

Compute Intensive Processes

Create a set of processes that do not request I/O or synchronization services. These are just fillers to take up time. Dynamically create and terminate with no more than 25 running at a time. These are essentially like the processes simulated in Problem 3.

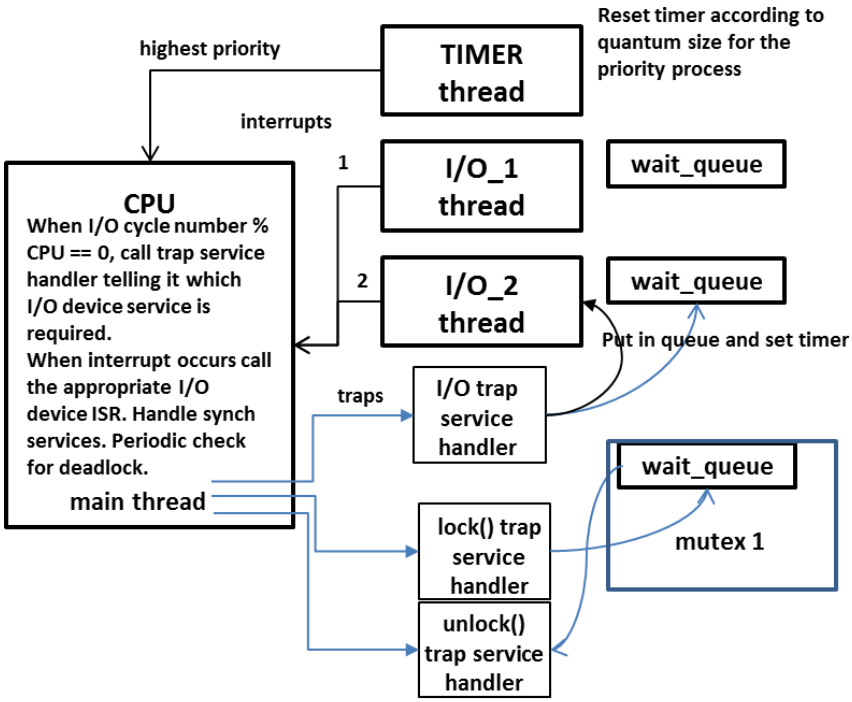

Figure 1. *Now the timer, and I/O devices (along with the main CPU) will be implemented as pthreads.*

Communicating Processes

Producer-Consumer Pairs

You will need to set up some process pairs (no more than ten). In each pair one process will be a producer and the other will be a consumer. These processes are just like all the others except that they are created in pairs and have some additional functions to handle their special jobs.

To allow the pair to communicate you will need a global integer that can be read from and written to by the pair. This variable needs to be protected in a critical region.

The pair will be cooperating to increment a single integer variable, initialized to zero when the mutex is created. The producer will increment a variable each time through its cycle. The consumer will read the variable and print it out in the trace when it runs. You will get the producer-consumer model in class.

In your traces you should see your producer process write the sequence of numbers (i.e. 1, 2, 3,...n) to the shared space and the consumer reading the sequence (and print to the trace to verify) in the same order. No duplicate or missed reading events should occur.

I would recommend you label your producers and consumers (in pairs) with names and not just PIDs. That way you can more easily pick out paired operations. E.g., the trace would read:

Producer X incremented variable XY: 313
[some time later]
:
Consumer Y read variable XY: 313

Mutual Resource Users

These processes should also be created in pairs but there will be two versions to be used in experiments on deadlock. The objective is to get two or more processes that attempt to acquire the same resources. In one version we will simulate the correct situation of no circular waiting to prevent deadlocks. In the other run we will set up the locking steps so that we permit all four conditions for deadlock. There is no guarantee that deadlock will occur but over several runs of the simulation you should see some processes in deadlock. To detect deadlock you will write a special monitor that will run every once in a while to check the waiting queues to see if circular waiting is obtained. See below.

For no deadlock your processes should work as follows:

Process A type	**Process B type**
lock mutex R1 (resource 1)	lock mutex R1
lock mutex R2 (resource 2)	lock mutex R2
print to trace that both resources are used	print to trace that both resources are used
unlock mutex R2	unlock mutex R2
unlock mutex R1	unlock mutex R1
proceed around the loop	proceed around the loop

To obtain the possibility for deadlock Process B types should reverse the order of locking and unlocking, i.e. lock R2 first then R1 and unlock R1 and then R2. With enough of these types of processes in the mix you should be able to get one instance of deadlock.

IMPLEMENTING SYNCHRONIZATION SERVICES

You will be simulating synchronization services in order to implement coordination between various processes. Specifically you will implement mutexes and condition variables based on the pthread versions (as in the book). DO NOT USE THE PTHREAD FUNCTIONS in the pthreads library except to coordinate the CPU, timer, and I/O devices threads. Your task is to write simulations of these functions for coordinating your simulated processes.

Each synchronization tool will be built using a pointer variable to the process that currently "owns" the mechanism. For example a mutex struct would contain a pointer variable to the process that has a lock on the mutex. The struct will also contain a FIFO-Queue to hold process pointers that are currently blocked awaiting getting a lock.

You will need to build several more arrays (like the I/O arrays) into the kinds of processes (above) that will be using synchronization tools. These arrays will contain the step numbers when these processes execute traps to **lock**, or **trylock**, or **unlock** a mutex. They will also contain steps when processes **wait** or **signal** condition variables. The difference between these arrays and the I/O arrays is that these arrays will be hand-crafted to implement the appropriate situations as presented above. You will hard-code the values into the arrays but you must also alter your random generating algorithm for the I/O arrays to make sure none of the numbers recorded there interfere with these hand-crafted numbers, i.e. you don't want to trap to an I/O after locking a mutex!

Traces

Use the same format for recording events in the runs as you did in the last problem assignment. This time add the new events, i.e.

PID xxxx: requested lock on mutex Mx - blocked by PID yyyy
PID zzzz: requested lock on mutex Mz - succeeded
PID aaaa requested condition wait on cond aaaa with mutex bbbb
PID bbbb sent signal on cond aaaa

Make sure there is enough information in these traces that you can check the correctness of the synchronization pairs given above.

SPECIAL DEADLOCK MONITOR

This function is not part of a regular scheduler but for purposes of simulation we often write special monitors or *instruments* to measure particular attributes of the simulation. In this case you will write a function that is called periodically (say every tenth context switch) and will determine if processes in the waiting queues for R1 and R2 are stuck due to circular waiting. I leave it to your ingenuity to figure out how you can make this detection (the book has some clues; see chapter 32).

During the runs that are set up to NOT have deadlock your monitor should report at the end that no deadlocks were detected. If it reports that deadlock did occur then you will have some debugging to do. These reports can be treated like events and printed to the output file.

During the runs that are set up so that deadlock might occur your monitor should report each time it is called (to the output file) the same message (e.g. "no deadlock detected"). But if a deadlock is detected it should print out something like, "deadlock detected for processes PIDxxxx and PIDyyyy".

Experiments

Set up your simulation to go through a fixed number of loops, say 100,000 for each run. Be sure to seed the random number generator with a different number each time (a convenient way to do this is to use the system clock at runtime) you do a run.

Run the simulator at least ten times with the no-deadlock and ten times with the deadlock possible setups. Do a search through the output files for key events (deadlock in particular) and summarize these in a report.

REPORT

An additional deliverable for this project is the Final Report. You will need to do analysis on the data that your simulations generated.

For each run of the simulator collect data on the number of processes that were run (total) and the numbers of processes still in all of the queues (ready and waiting) at the termination of the run. This will require an additional kind of "instrumentation". Collect all of these data for all runs and create a text report that summarizes the data. Answer these questions:

- How many total processes were created?
- How many of each kind were created? How does this comport with your algorithm for creating relative numbers of each kind?
- How many were terminated normally (only deadlock situations should result in abnormal termination)?
- How many were in each priority queue at the end of the run?
- In the deadlock experiments, how many, on average, were experienced?
- If there were parts of the requirements that you were unable to complete, explain why, e.g., if you could not solve a bug or just didn't have time.

THREADED PROGRAMS

In using pthreads, 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 in Problem 4.

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 (Links to an external site.)Links to an external site.). 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/

TURN-IN:

Zip up: Source code files, one text result (trace) file (or part of the file) from each of the no-deadlock and the deadlock-possible runs, and the report file. One person per team submit here.