1 Overview

In this lab, you will implement a multiprocessor operating system simulator using a popular userspace threading library for linux called pthreads. The framework for the multithreaded OS simulator is nearly complete, but missing one critical component: the CPU scheduler! Your task is to implement the CPU scheduler, using three different scheduling algorithms.

We have provided you with source files that constitute the framework for your simulator. You will only need to modify student.c. However, just because you are only modifying two files doesn't mean that you should ignore the other ones - there is helpful information in the other files. We have provided you these files:

1. Makefile - Working one provided for you; do not modify.
2. os-sim.c - Code for the operating system simulator which calls your CPU scheduler.
3. os-sim.h - Header file for the simulator.
4. process.c - Descriptions of the simulated processes.
5. process.h - Header file for the process data.
6. student.c - This file contains stub functions for your CPU scheduler.
7. student.h - Header file for your code to interface with the OS simulator

1.1 Scheduling Algorithms

For your simulator, you will implement the following three CPU scheduling algorithms:

1. First Come, First Serve (FCFS) - Runnable processes are kept in a ready queue. FCFS is non-preemptive; once a process begins running on a CPU, it will continue running until it either completes or blocks for I/O.
2. Preemptive Priority – Each process is assigned a priority. You must schedule the processes accordingly. If a process that joins the ready queue has a higher priority than a process that is currently running, you must preempt the currently running process and schedule the new process.
3. Shortest Job First (SJF) – Always schedules the process with the shortest initial run time first. SJF is not preemptive.

1.2 Process States

In our OS simulation, there are five possible states for a process, which are listed in the process state t enum in os-sim.h:

1. NEW - The process is being created, and has not yet begun executing.
2. READY - The process is ready to execute, and is waiting to be scheduled on a CPU.
3. RUNNING - The process is currently executing on a CPU.
4. WAITING - The process has temporarily stopped executing, and is waiting on an I/O request to complete.
5. TERMINATED - The process has completed.

There is a field named state in the PCB, which must be updated with the current state of the process. The simulator will use this field to collect statistics.

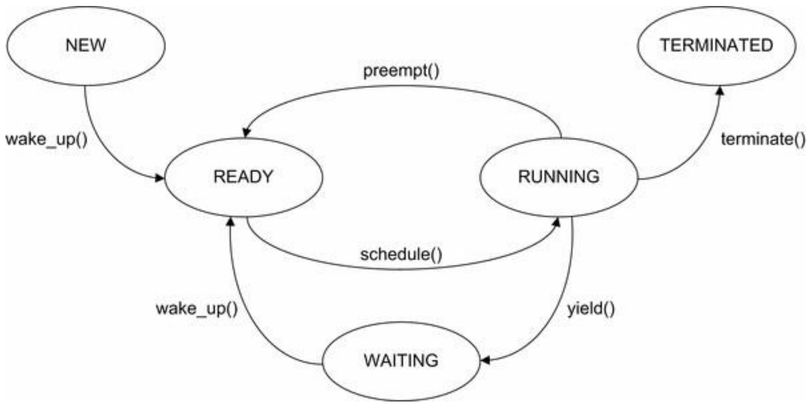


Figure 1: Process States

1.3 The Ready Queue

On most systems, there are a large number of processes, but only one or two CPUs on which to execute them. When there are more processes ready to execute than CPUs, processes must wait in the READY state until a CPU becomes available. To keep track of the processes waiting to execute, we keep a ready queue of the processes in the READY state

Since the ready queue is accessed by multiple processors, which may add and remove processes from the ready queue, the ready queue must be protected by some form of synchronization–for this lab assignment, you will use a mutex lock. The ready queue SHOULD use a different mutex than the running processes mutex.

1.4 Scheduling Processes

schedule() is the core function of the CPU scheduler. It is invoked whenever a CPU becomes available for running a process. schedule() must search the ready queue, select a runnable process, and call the context switch() function to switch the process onto the CPU.

There is a special process, the idle process, which is scheduled whenever there are no processes in the READY state.

1.5 CPU Scheduler Invocation

There are four events which will cause the simulator to invoke schedule():

1. yield() - A process completes its CPU operations and yields the processor to perform an I/O request.

2. wake up() - A process that previously yielded completes its I/O request, and is ready to perform CPU operations. wake up() is also called when a process in the NEW state becomes runnable.

3. preempt() - When using a Preemptive Priority scheduling algorithm, a CPU-bound process may be preempted before it completes its CPU operations.

4. terminate() - A process exits or is killed.

The CPU scheduler also contains one other important function: idle(). idle() contains the code that gets by the idle process. In the real world, the idle process puts the processor in a low-power mode and waits. For our OS simulation, you will use a pthread condition variable to block the thread until a process enters the ready queue.

1.6 The Simulator

We will use pthreads to simulate an operating system on a multiprocessor computer. We will use one thread per CPU and one thread as a 'supervisor' for our simulation. The CPU threads will simulate the currently- running processes on each CPU, and the supervisor thread will print output and dispatch events to the CPU threads.

Since the code you write will be called from multiple threads, the CPU scheduler you write must be thread- safe! This means that all data structures you use, including your ready queue, must be protected using mutexes.

The number of CPUs is specified as a command-line parameter to the simulator. For this project, you will be performing experiments with 1, 2, and 4 CPU simulations.

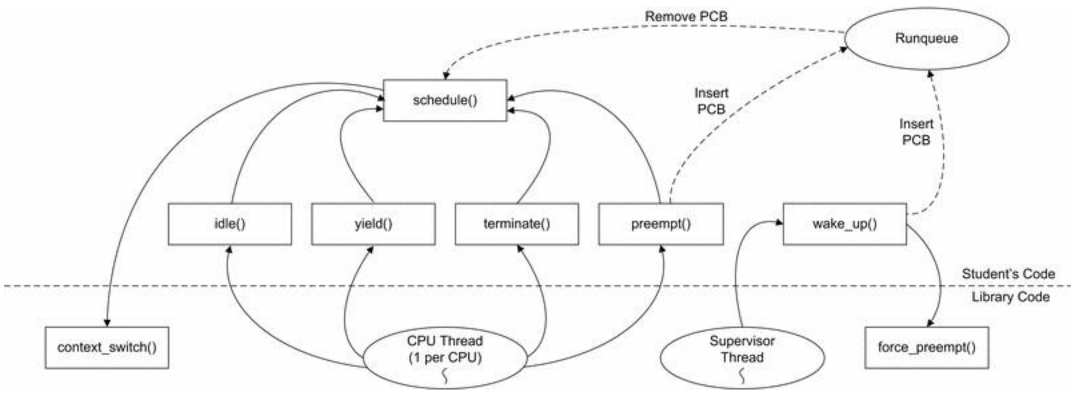
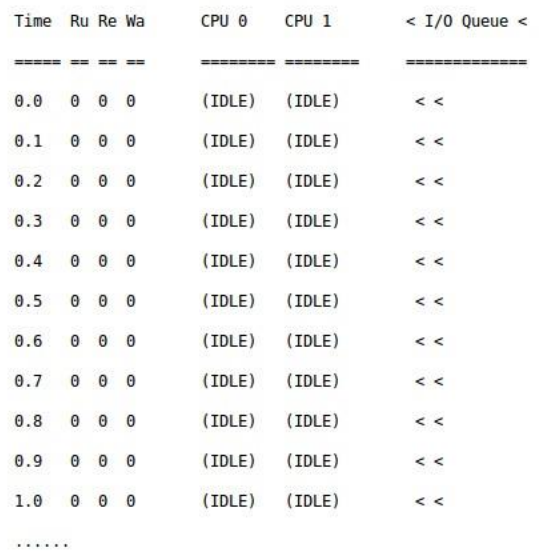


Figure 2: Simulator Function Calls

After parsing the input arguments, compile and run the simulator with ./os-sim 2. After a few seconds, hit Control-C to exit. You will see the output below:



```
Time  Ru Re Wa     CPU 0    CPU 1       < I/O Queue <
===== == == ==     ======== ========    ============
0.0   0  0  0      (IDLE)   (IDLE)       < <
0.1   0  0  0      (IDLE)   (IDLE)       < <
0.2   0  0  0      (IDLE)   (IDLE)       < <
0.3   0  0  0      (IDLE)   (IDLE)       < <
0.4   0  0  0      (IDLE)   (IDLE)       < <
0.5   0  0  0      (IDLE)   (IDLE)       < <
0.6   0  0  0      (IDLE)   (IDLE)       < <
0.7   0  0  0      (IDLE)   (IDLE)       < <
0.8   0  0  0      (IDLE)   (IDLE)       < <
0.9   0  0  0      (IDLE)   (IDLE)       < <
1.0   0  0  0      (IDLE)   (IDLE)       < <
......
```

Figure 3: Sample Output

The simulator generates a Gantt Chart, showing the current state of the OS at every 100ms interval. The leftmost column shows the current time, in seconds. The next three columns show the number of Running, Ready, and Waiting processes, respectively. The next two columns show the process currently running on each CPU. The rightmost column shows the processes which are currently in the I/O queue, with the head of the queue on the left and the tail of the queue on the right.

As you can see, nothing is executing. This is because we have no CPU scheduler to select processes to execute! Once you complete Problem 1 and implement a basic FCFS scheduler, you will see the processes executing on the CPUs.

2 Implementation [100 points + 20 points extra credit]

Main

You must parse the command line arguments to run the program accordingly.
To run FCFS, type "./os-sim –f <#CPUs>
To run Preemptive Priority, type "./os-sim –p <#CPUs>
To run SJF, type "./os-sim –s <#CPUs>"

FCFS Scheduler [50 points]

Implement the CPU scheduler using the FCFS scheduling algorithm. You may do this however you like, however, we suggest the following:

• Implement a thread-safe ready queue using a linked list.

• Implement the yield(), wake up(), and terminate() handlers. preempt() is not necessary for this stage of the project. See the overview and the comments in the code for the proper behavior of these events.

• Implement idle(). idle() must wait on a condition variable that is signaled whenever a process is added to the ready queue.

• Implement schedule(). schedule() should extract the first process in the ready queue, then call con- text switch() to select the process to execute. If there are no runnable processes, schedule() should call context switch() with a NULL pointer as the PCB to execute the idle process.

2.1 Hints

- Be sure to update the state field of the PCB. The library will read this field to generate the Running, Ready, and Waiting columns, and to generate the statistics at the end of the simulation.
- There is a field in the PCB, next, which you may use to build linked lists of PCBs.
- Four of the five entry points into the scheduler (idle(), yield(), terminate(), and preempt()) should cause a new process to be scheduled on the CPU. In your handlers, be sure to call schedule(), which will select a runnable process, and then call context switch(). When these four functions return, the library will simulate the process selected by context switch().

Preemptive Priority [50 points]

Add Preemptive Priority scheduling to your code

The scheduler should use the priority field of the PCB to select which process to run

For Preemptive Priority scheduling, you will need to make use of the running_processes[] array and force preempt() function. The running_processes[] array should be used to keep track of the process currently executing on each CPU. Since this array is accessed by multiple CPU threads, it must be protected by a mutex. Running processes mutex has been provided for you.

Extra Credit: SJF [20 points]

Add SJF scheduling functionality to your code
You may modify the other files given to you to implement SJF

3 Deliverables

Failure to follow submission instructions will result in a 0 on the assignment.
Do not modify any file other than Student.c for FCFS and Preemptive Priority.
If you do the extra credit, please submit it separately and make sure that your other files are not modified when you submit FCFS and Preemptive Priority.
Do not change the names of any of the variables that we provide for you
Do not submit files with extra print statements

Type ‘make submit’ to generate a tarball containing all the files needed for submission. Please turn in the tarball that is generated.

This tarball should contain:

- Makefile - Working one provided for you; don’t break it.
- src/os-sim.c - Code for the operating system simulator.
- src/os-sim.h - Header file for the simulator.
- src/process.c - Descriptions of the simulated processes.
- src/process.h - Header file for the process data.
- src/student.c - Your code for the scheduler.
- src/student.h - Header file for your scheduler code.