1 Overview

In this project, 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.

Note: Make sure that multiple CPU cores are enabled in your virtual machine, otherwise you will receive incorrect results. See the TAs if you need help.

If you are using the CS 2200 Vagrant box, the number of cores should default to 2. You can run `nproc --all` to see how many cores are available to your VM.

We have provided you with source files that constitute the framework for your simulator. You will only need to modify `answers.txt` and `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. Also contains ready queue struct definition.

1.1 Scheduling Algorithms

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

1. **First In, First Out (FIFO)** - Runnable processes are kept in a ready queue. FIFO 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. **Round-Robin** - Similar to FIFO, except preemptive. Each process is assigned a timeslice when it is scheduled. At the end of the timeslice, if the process is still running, the process is preempted, and moved to the tail of the ready queue.
3. **Priority with Preemption** - Each process has a priority associatted with it. Processes with higher priority will be scheduled first. The process with the highest priority always gets the CPU. Processes with lower priority must be preempted if a process with a higher priority becomes runnable.

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.

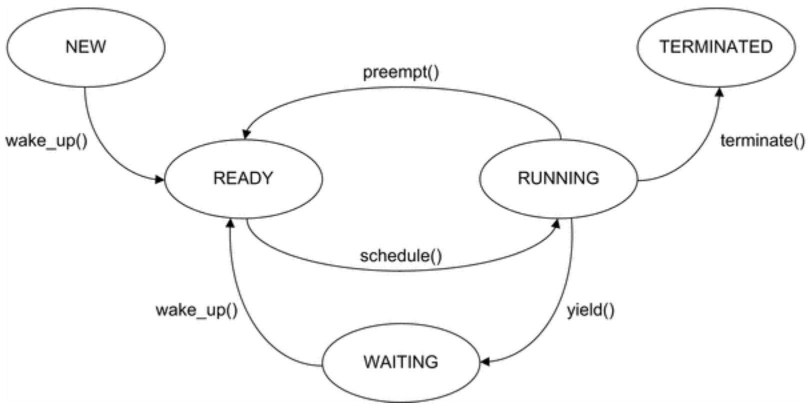


Figure 1: Process States

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 project, you will use a mutex lock that we have provided called `ready_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 Round-Robin or 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.

Also, for demonstration purposes, the simulator executes much slower than a real system would. In the real world, a CPU burst might range from one to a few hundred milliseconds, whereas in this simulator, they range from 0.2 to 2.0 seconds.

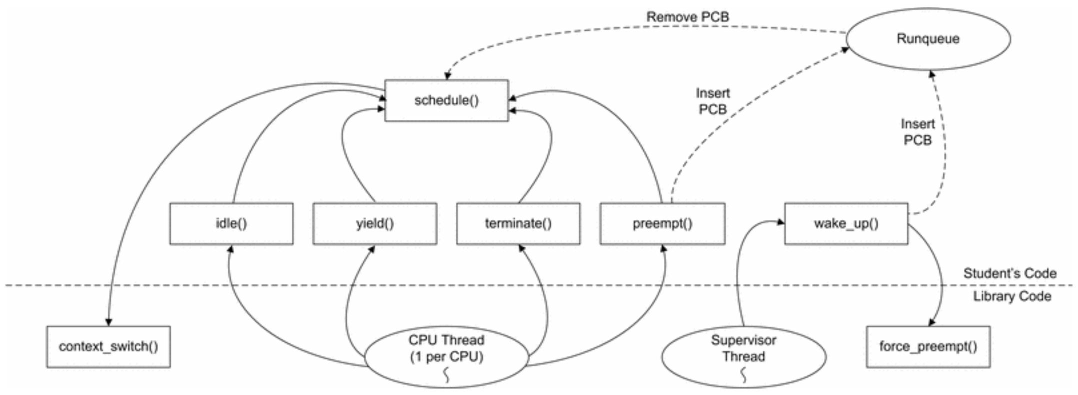
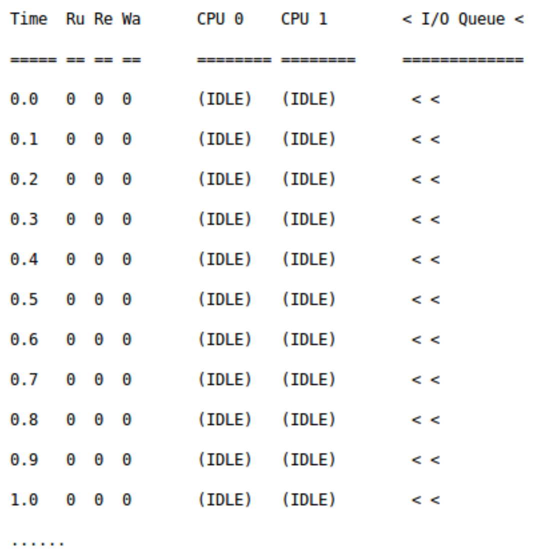


Figure 2: Simulator Function Calls

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 FIFO scheduler, you will see the processes executing on the CPUs.

2 Problem 0: The Ready Queue

Implement the helper functions `enqueue()`, `dequeue()`, and `is_empty()` for the queue struct provided. The struct will serve as your ready queue, and you should be using these helper functions to add and remove processes from the ready queue in the problems to follow. You can find the declarations of `queue_t`, `enqueue()`, `dequeue()`, and `is_empty()` in `student.h`.

2.1 Hints

- Your queue should be backed by a linked list with each PCB acting as a node. There is a field in the PCB, `next`, which you may use to build linked lists of PCBs.
- You should not be editing any of the fields of the PCBs inside the queue aside from `next`.
- There is an edge case for `dequeue()` which you must handle. Take a look at the documentation for the function for more details.
- When using the ready queue helper functions in the following problems, make sure to call them in a thread-safe manner. Read up on how to use mutex locks and lock/unlock your queue struct if and when you call these functions.

3 Problem 1: FIFO Scheduler

NOTE: Part B of this and each following problem requires you to put your answer down in answers.txt

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

- 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 signalled whenever a process is added to the ready queue.
- Implement `schedule()`. `schedule()` should extract the first process in the ready queue, then call `context_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.

3.1 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.
- 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()`.
- `context_switch()` takes a timeslice parameter, which is used for preemptive scheduling algorithms. Since FIFO is non-preemptive, use -1 for this parameter to give the process an infinite timeslice.
- Make sure to use the helper functions in a thread-safe manner when adding and removing processes from the ready queue!

Part B. Run your OS simulation with 1, 2, and 4 CPUs. Compare the total execution time of each. Is there a linear relationship between the number of CPUs and total execution time? Why or why not? Keep in mind that the execution time refers to the simulated execution time.

4 Problem 2: Round-Robin Scheduler

Part A. Add Round-Robin scheduling functionality to your code. You should modify main() to add a command line option, -r, which selects the Round-Robin scheduling algorithm, and accepts a parameter, the length of the timeslice. For this project, timeslices are measured in tenths of seconds. E.g.:

```
./os-sim <# CPUs> -r 5
```

should run a Round-Robin scheduler with timeslices of 500 ms. While:

```
./os-sim <# of CPUs>
```

should continue to run a FIFO scheduler. You should also make sure preempt is implemented in this section of the project.

To specify a timeslice when scheduling a process, use the timeslice parameter of `context_switch()`. The simulator will automatically preempt the process and call your `preempt()` handler if the process executes on the CPU for the length of the timeslice without terminating or yielding for I/O.

Part B. Run your Round-Robin scheduler with timeslices of 800ms, 600ms, 400ms, and 200ms. Use only one CPU for your tests. Compare the statistics at the end of the simulation. Show that the total waiting time decreases with shorter timeslices. However, in a real OS, the shortest timeslice possible is usually not the best choice. Why not?

5 Problem 3: Priority with Preemption

Part A. Add priority with preemption scheduling to your code. Modify `main()` to accept the -p parameter to select the priority algorithm. The -r and default FIFO scheduler should continue to work. The scheduler should use the `priority` field of the PCB to prioritize processes that have a higher priority (**NOTE: Lower number means higher priority**).

For priority scheduling, you will need to make use of the `current[]` array and `force_preempt()` function. The `current[]` 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. `current_mutex` has been provided for you. The `force_preempt()` function preempts a running process before it's finished executing. Your `wake_up()` handler should make use of this function to preempt a process when a process with a higher priority needs a CPU.

Part B. Priority schedulers can sometimes lead to starvation among processes with lower priority. What is a way that operating systems can mitigate starvation in a priority scheduler?

Run each of the scheduling algorithms using one CPU and compare the total waiting times. Which one had the lowest? Why?

6 Deliverables

NOTE: Each problem (excluding 0) has two parts (labeled A and B). The first is the actual implementation, and the second is a question linked to the scheduling algorithm you are implementing. Make sure you complete both.

You can run `make submit` to automatically package your project for submission. Submit the resulting **tar.gz** compressed folder on Canvas.

The archive should contain the following files:

- `answers.txt`- Short answers to questions from above.
- `Makefile` - Working one provided for you; don't break it.
- `os-sim.c` - Code for the operating system simulator.
- `os-sim.h` - Header file for the simulator.
- `process.c` - Descriptions of the simulated processes.
- `process.h` - Header file for the process data.
- `student.c` - Your code for the scheduler.
- `student.h` - Header file for your scheduler code.

Keep your answers detailed enough to cover the question, including support from simulator results if appropriate. Don't write a book; but if you're not sure about an answer, err on the side of giving us too much information.