

# Project 4: Process Scheduling Simulation

## 1 Overview

In this project, you will implement a multiprocessor operating system simulator using a popular threading library for linux called `pthread`s. The framework for the multiprocessor OS simulator is nearly complete, but missing one critical component: the process scheduler! Your task is to implement the process scheduler and three different scheduling algorithms.

The simulated operating system supports only one thread per process making it similar to the systems that we discussed in Chapter 6. However, the simulator itself will use a thread to represent each of the CPUs in the simulated hardware. This means that the CPUs in the simulator will appear to operate concurrently.

**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 VirtualBox or the CS 2200 Vagrant box, the number of cores should default to 2. In Vagrant, 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. We have provided you the following files:

1. `os-sim.c` - Code for the operating system simulator which calls your CPU scheduler.
2. `os-sim.h` - Header file for the simulator.
3. `process.c` - Descriptions of the simulated processes.
4. `process.h` - Header file for the process data.
5. `student.c` - This file contains stub functions for your CPU scheduler.
6. `student.h` - Header file for your code to interface with the OS simulator. Also contains the ready queue struct definition.

**Reminder: The only files that you need to edit are `student.c` and `answers.txt`. If you edit any other files, your code may fail the autograder!**

### 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. **Round-Robin** - Similar to FCFS, 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. **Preemptive Priority (PS)** - Each process has a priority associated with it. Processes with higher priority will be scheduled first. Note that we represent the priority as an integer and smaller integers have higher priority. For example, a process with priority of 1 will be scheduled before a process with priority 4. Processes with the same priority will be scheduled in the same manner as FCFS. Since this

algorithm is preemptive, when a process with higher priority (lower int value) wakes up, you should preempt the currently running process with the lowest priority.

## 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.

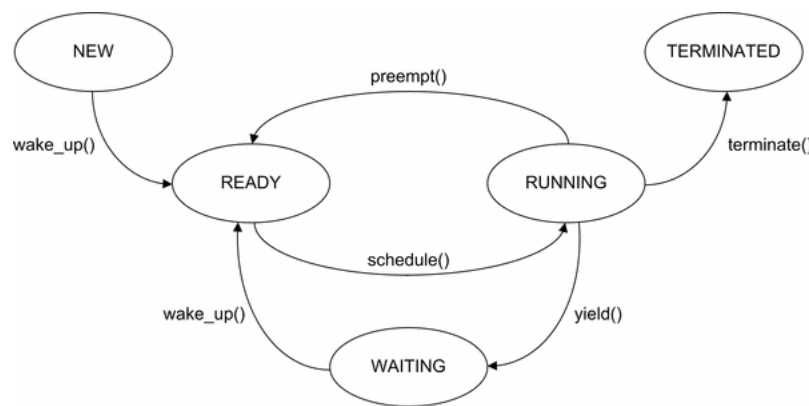


Figure 1: Process States

## 1.3 The Ready Queue

On most systems, there are a large number of processes that need to share the resources of one or two CPUs. 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.

Note that in a multiprocessor environment, we cannot mandate that the currently running process be at the head of the ready q. There is an array (one entry for each cpu) that will hold the pointer to the PCB currently running on that cpu.

There is a special process, the idle process, which is scheduled whenever there are no processes in the READY state. This process simply waits for something new to be added to the ready queue and then calls `schedule()`.

## 1.5 CPU Scheduler Invocation

There are five 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 scheduling algorithm, a CPU-bound process may be preempted before it completes its CPU operations.
4. `terminate()` - A process exits or is killed.
5. `idle()` - Waits for a new process to be added to the ready queue. `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 supervisor thread will spawn new processes (as if a user started a process). The CPU threads will simulate the currently-running processes on each CPU, and the supervisor thread will print output.

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.

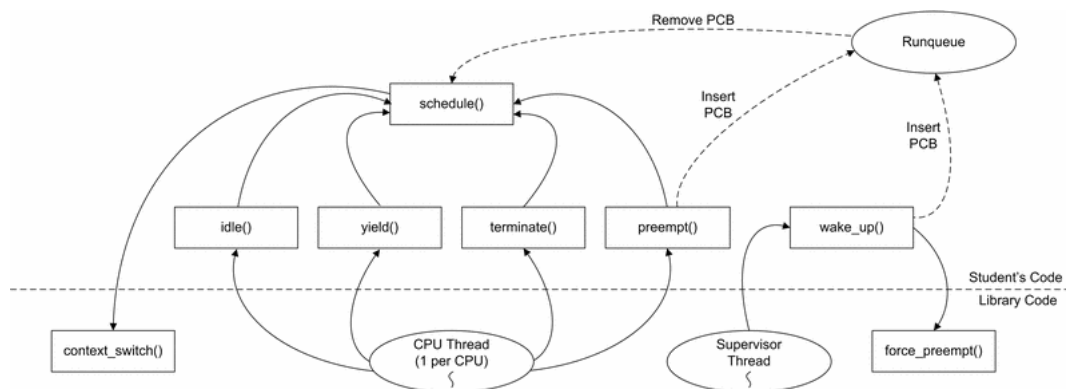


Figure 2: Simulator Function Calls

The above diagram should give you a good overview of how the system works in terms of the functions being called and PCBs moving around. Below is a second diagram that shows the entire system overview and the code that you need to write is inside of the green cloud at the bottom. All of the items outside of the green cloud are part of the simulator and will not need to be modified by you.

### USER/KERNEL DELINEATIONS FOR PROJECT 3

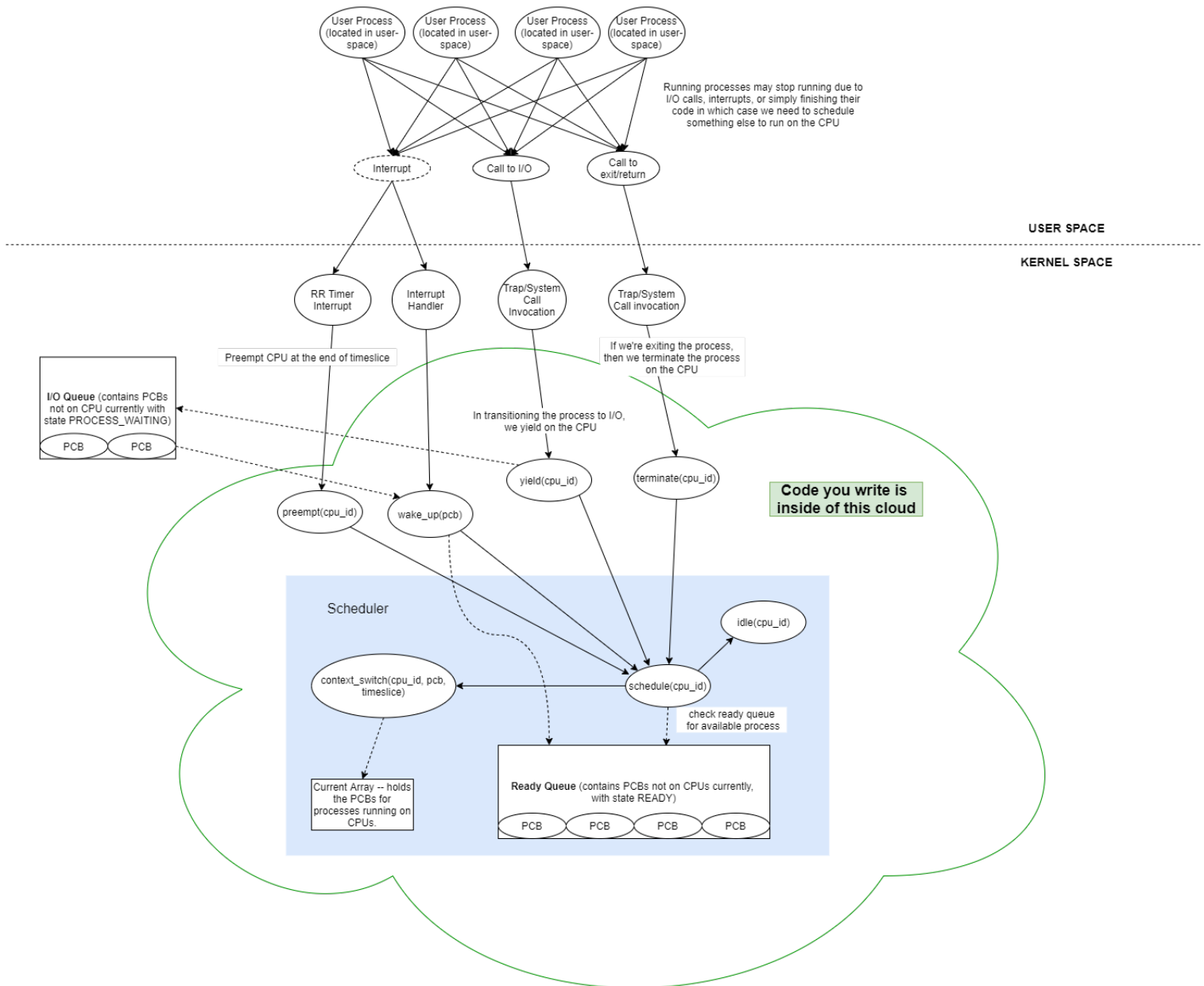


Figure 3: System overview

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 4: 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()` in `student.c`.**

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`.
- `enqueue()` should add a process to the ready queue at the appropriate location.
- `dequeue()` should remove a process at the head of the ready queue and return a pointer to that process. There is an edge case for `dequeue()` which you must handle. If the queue is empty, simply return `NULL`.
- 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: FCFS Scheduler

NOTE: Part C of this requires you to put your answer down in `answers.txt`

### 3.1 Part A

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

- Implement the `yield()`, `wake_up()`, and `terminate()` handlers. in `student.c`.

Checkout the hints in section 3.3, and note that `preempt()` is not necessary for this stage of the project.

- Implement `idle()`.

`idle()` must wait on a condition variable that is signalled whenever a process is added to the ready queue.

### 3.2 Part B

- 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.

### 3.3 Hints

- Be sure to update the `state` field of the PCB in the methods from part A and B. 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 execution of the process selected by `context_switch()`.
- `context_switch()` takes a timeslice parameter, which is used for preemptive scheduling algorithms. Since FCFS 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!
- 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.

### 3.4 Part C

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

NOTE: Part B of this requires you to put your answer down in `answers.txt`

## 4.1 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 FCFS scheduler. Note: you can use `getopt()`, which we used earlier in the semester or just parse the command line arguments passed into `main` using `if` statements.

### Implement `preempt()`.

`preempt()` should place the currently running process back in the ready queue, and call `schedule()` to select a new runnable process.

To specify a timeslice when scheduling a process, use the timeslice parameter of `context_switch()`. The simulator will simulate a timer interrupt to 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.

## 4.2 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. Is there a relationship between the total waiting time and timeslice length? If so, what is it? In contrast, in a real OS, the shortest timeslice possible is usually not the best choice. Why not?

# 5 Problem 3: Preemptive Priority

**NOTE: Part B of this requires you to put your answer down in `answers.txt`**

## 5.1 Part A.

Add priority scheduling to your code. You should change your `enqueue()` or `dequeue()` functions to make the ready queue a priority queue. The scheduler should use the `priority` field of the PCB to prioritize processes that have a higher priority. (**NOTE: Lower integer means higher priority**).

You should also modify the `wake_up()` function to preempt the process with the lowest priority currently running on the CPUs if the process that is waking up is a higher priority (lower int value) than any currently running process. If there's an idle CPU or if all the processes currently running have higher priority, you don't have to force preempt any process. To preempt a process, use `force_preempt()`. Look in `os-sim.h` for its prototype and the parameters it takes in.

Modify `main()` to accept the `-p` parameter to select PS as the `scheduler_algorithm`. The default FCFS scheduler should continue to work. (Hint: create a global variable to identify the type of your scheduling algorithm).

For priority scheduling, 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.

## 5.2 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?

## 6 Problem 4: The Priority Inversion Problem (Short Answer)

Consider a non-preemptive priority scheduler. Suppose you have a high priority process (P1) that wants to display a window on the monitor. But, the window manager is a process with low priority and will be placed on the end of the ready queue. While it is waiting to be scheduled, new medium priority processes are likely to come in and starve the window manager process. The starvation of the window manager will also mean the starvation of P1 (the process with high priority), since P1 is waiting for the window manager to finish running.

If we want to keep our non-preemptive priority scheduler, what edits can we make to our scheduler to ensure that the P1 can finish its execution before any of the medium priority processes finish their execution. Explain in detail the changes you would make.

## 7 The Gradescope Environment

You will be submitting files to Gradescope, where they will be tested in a VM setup that runs through Gradescope. The specifications of this VM are that it runs **Ubuntu 20.04 LTS (64-bit)** and **gcc 9.3.0**, and so we expect that your files can run in such a setup. This means that *when you are running your project locally*, you will want to ensure you are using a VM/some setup that runs **Ubuntu 20.04 LTS (64-bit)** and **gcc 9.3.0**; this way, you can ensure that what occurs locally is what will occur when you submit to Gradescope.

**IMPORTANT:** Since we are dealing with different threads of execution, the result of each run in the simulation will be different. As a result, our gradescope autograder will accept a range of results for your simulation. In past semesters, we found that the range will start to **increase** the more computations you do in your enqueue/dequeue methods. If you find that you aren't passing the autograder but you believe that your code is right, it is likely that you need to trim down your enqueue/dequeue methods. We are planning to start our autograder with a tighter acceptable range, and if required, we will increase it.

## 8 How to Run / Debug Your Code

We have provided a Makefile that will run gcc for you. To compile your code with no optimizations (which you should do while developing, it will make debugging easier) and test with the fcfs algorithm and one CPU, run:

```
$ make debug
$ ./os-sim 1
```

To run the other algorithms, run with the flags you implemented for round robin and priority. Remember that round robin requires you to enter a time slice.

## 9 Deliverables

**NOTE:** Each problem (excluding Problem 0 and Problem 4) has both a coding and short response part. Make sure you complete both.

You need to upload **student.c**, and **answers.txt** to Gradescope, and an autograder will run to check if your scheduler is working. The autograder might take a couple of minutes to run. Remember to upload **student.c**, and **answers.txt** for every submission as your lastest submission would be one we will grade.



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.