

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 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 - 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.

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. **Shortest Job First (SJF)** - The SJF scheduler looks at the CPU burst times needed by the current set of processes that are ready to run and picks the one that needs the shortest CPU burst. SJF is also non-preemptive.
3. **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.
4. **EXTRA CREDIT: Shortest Remaining Time First (SRTF)** - The process with the shortest remaining time in its burst always gets the CPU. Longer processes must be pre-empted if a process that has a shorter burst becomes runnable. Implementing this algorithm is optional and correct execution will be given 5 extra points on this project.

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.

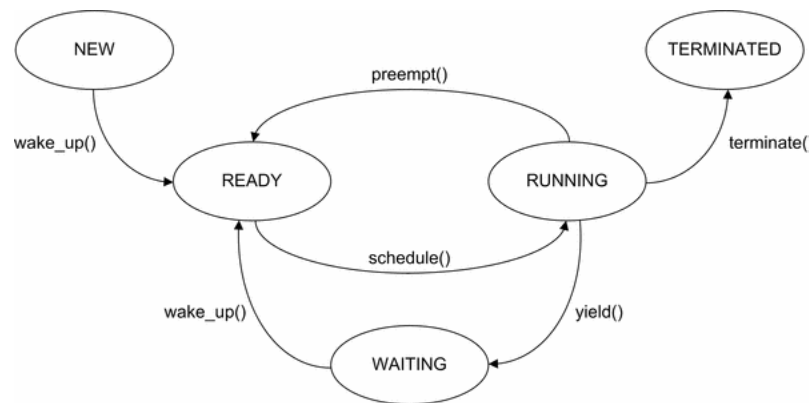


Figure 1: Process States

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.

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 (details below).

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 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.

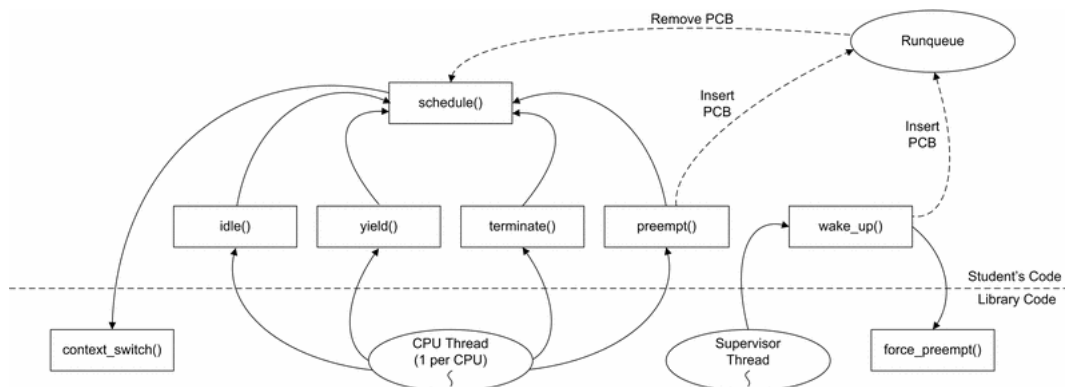


Figure 2: Simulator Function Calls

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. If you would like to zoom in to get a better look, this image is included in the project files as `system-diagram.jpg`.

USER/KERNEL DELINEATIONS FOR PROJECT 3

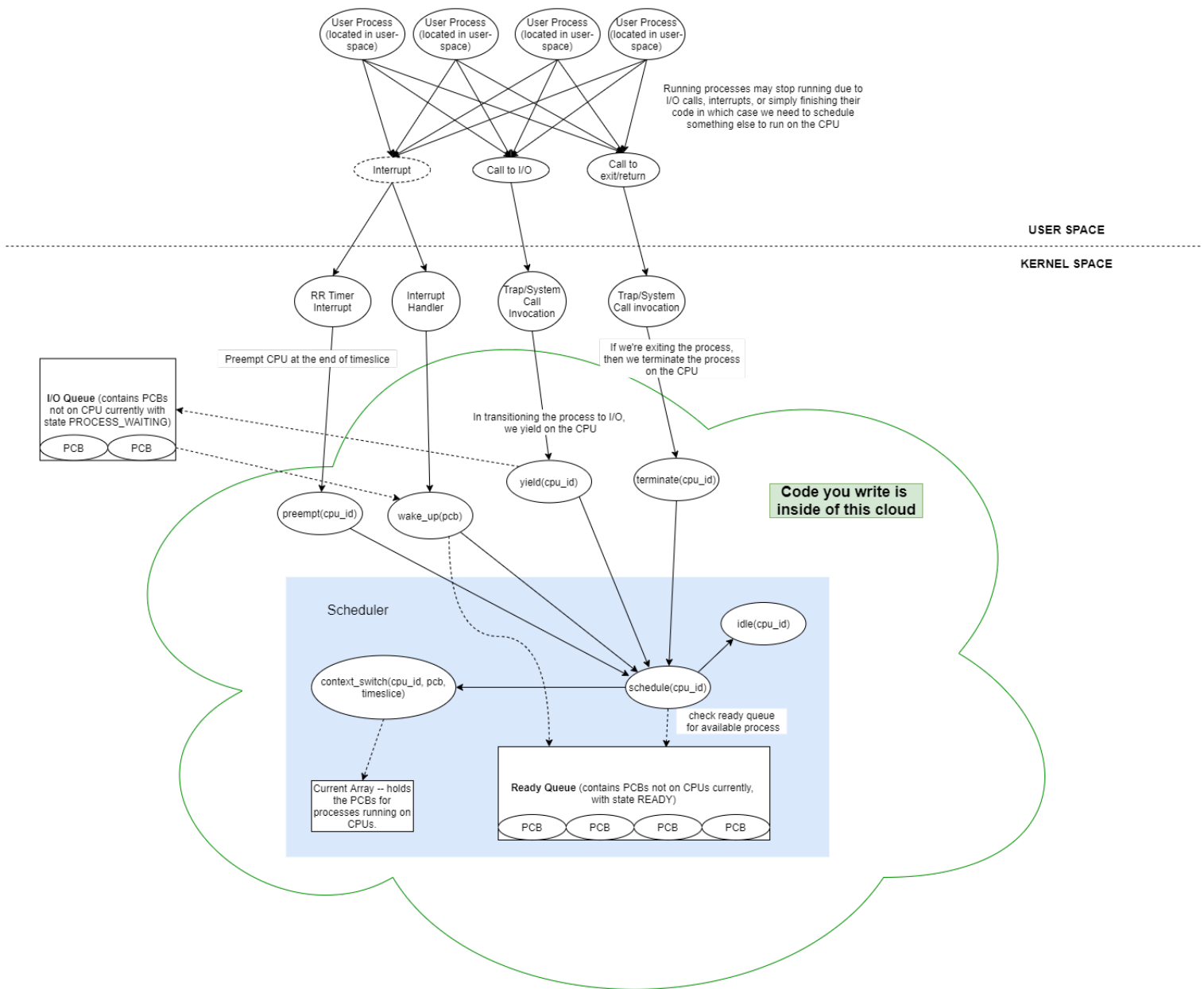


Figure 3: System overview

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` 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: FCFS 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 FCFS 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 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.

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: SJF Scheduler

Part A. Add a Shortest Job First scheduling functionality to your code.

First, modify `main()` to accept the `-j` parameter to select the SJF algorithm. The default FIFO scheduler should continue to work. The scheduler should use the `time_remaining` field of the PCB to prioritize processes that have a shorter remaining time in their CPU burst.

The job with the current shortest `remaining_time` will be executed and if there are at least two jobs in the ready queue with the same amount of CPU burst time remaining, the job that arrived first to the ready queue shall be run.

Since SJF is non-preemptive, all processes need to wait for the current process to give up the processor on their own accord. As in FCFS, the `current[]` array should be used to keep track of the process currently executing on each CPU and this array must be protected by a mutex.

Part B. While it is easy to simulate an SJF algorithm in the simulator, it is essentially impossible to implement precisely in real life and is thus usually replaced with a Priority algorithm. Why is this the case?

5 Problem 3: 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 FCFS 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 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.

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?

6 Problem 4: SRTF Scheduler

EXTRA CREDIT: To reiterate, implementing the SRTF Scheduler is entirely optional and there will be no penalty for absence of this scheduler's realization. A fully correct implementation however will be awarded extra credit points.

Note: There is no Part B for this problem.

Add SRTF scheduling to your code. Modify `main()` to accept the `-s` parameter to select the SRTF algorithm. The `-r`, `-j`, and default FIFO scheduler should continue to work. The scheduler should use the `time remaining` field of the PCB to prioritize processes that have a shorter remaining time in their CPU burst. For SRTF 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 its timeslice expires. Your `wake_up()` handler should make use of this function to preempt a process when a process with lower time remaining needs a CPU.

7 Problem 5: The Priority Inversion Problem (Short Answer)

Assume we have three processes P1, P2, and P3; P1 has high priority, P2 has medium priority, and P3 has low priority.

Assume P1 and P3 work with/access a shared resource S, and that P2 does not need or use S. There exists a locking mechanism so that for any process P which is currently using S, no other process can use S unless P gives up the lock.

Assume that, currently, P3 has control of S. If P1 tries to get access to S, P1 must wait until P3 (a lower priority process) gives up S (and releases the locking mechanism).

If P2 becomes runnable in this time interval, during the time that P1 is waiting on P3 to give up S, P2 will preempt P3, as P2 has higher priority than P3 and doesn't require the usage/access to S. As a result, P3 is unable to give up access to S (because the preemption done by P2 didn't affect P3's holding of the lock).

We face an interesting problem: we observe starvation of a high priority process P1 by lower priority processes P2 and P3. This is known as a *priority inversion problem*.

Assume we have a scheduler that does have preemption (and we cannot use non-preemptive schedulers). Assume also that we can decrease or increase the priority of a process *during its runtime on the CPU*. Given these constraints, how might we ensure that P1 is able to execute before P2 ?

8 The Gradescope Environment

You will be submitting files to Gradescope, where they will be tested on/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.

9 Deliverables

NOTE: Each problem (excluding Problem 0 and Problem 4) 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 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.

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.