Project 4 Implementing Processor Scheduler using pthreads

Thanks to Professor Ramachandran at Georgia Tech for the original project

---

Deliverables

You will complete this project with your new partner. Submit a zip folder with the 2 files student.c and answers.txt (or any other reasonable extension). Be sure to include BOTH PARTNER NAMES at the top of BOTH FILES.

Overview

In this project you will build a simulator of a simple multiprocessor operating system scheduler to learn more about operating systems in general and process schedulers in specific. You will do this in C using pthreads, to get more practice with synchronization as well. The provided 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.

I have provided you with the below source files. You will only need to modify answers.txt and student.c. However, there is helpful information in the other files; you should look through them.

- Makefile - simply use the command "make" to compile the simulator. Modify at your own risk.
- os-sim.c - Code for the operating system simulator which calls your CPU scheduler.
- 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 - This file contains stub functions for your CPU scheduler and functions to manipulate a basic FIFO ready queue.
- student.h - Header file for your code to interface with the OS simulator

Scheduling Algorithms

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

- *First-Come, First Served (FCFS)* - Runnable processes are kept in a first-in, first-out 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.

- *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.
- *Static Priority* - The processes with the highest priorities always get the CPU. Lower-priority processes may be preempted if a process with a higher priority becomes runnable.

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:

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

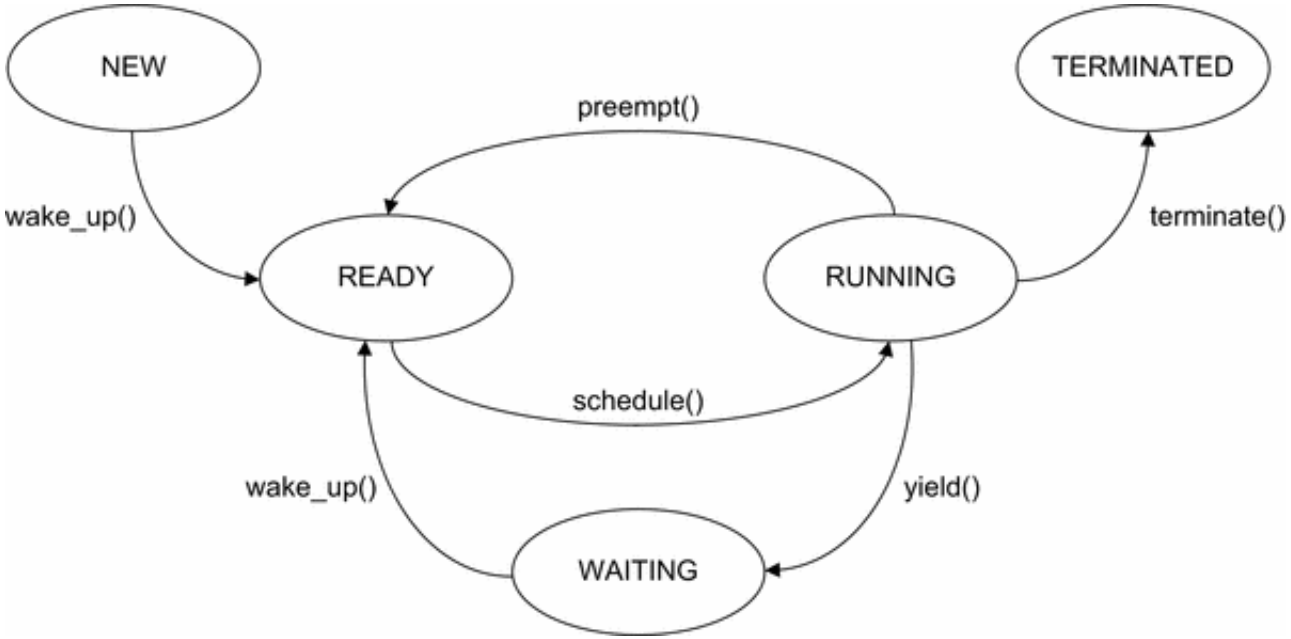

Figure 1: Process States

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, it will be a mutex lock.

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.

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 Static 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. In this OS simulation, it uses a pthread condition variable to block the thread until a process enters the ready queue.

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*.

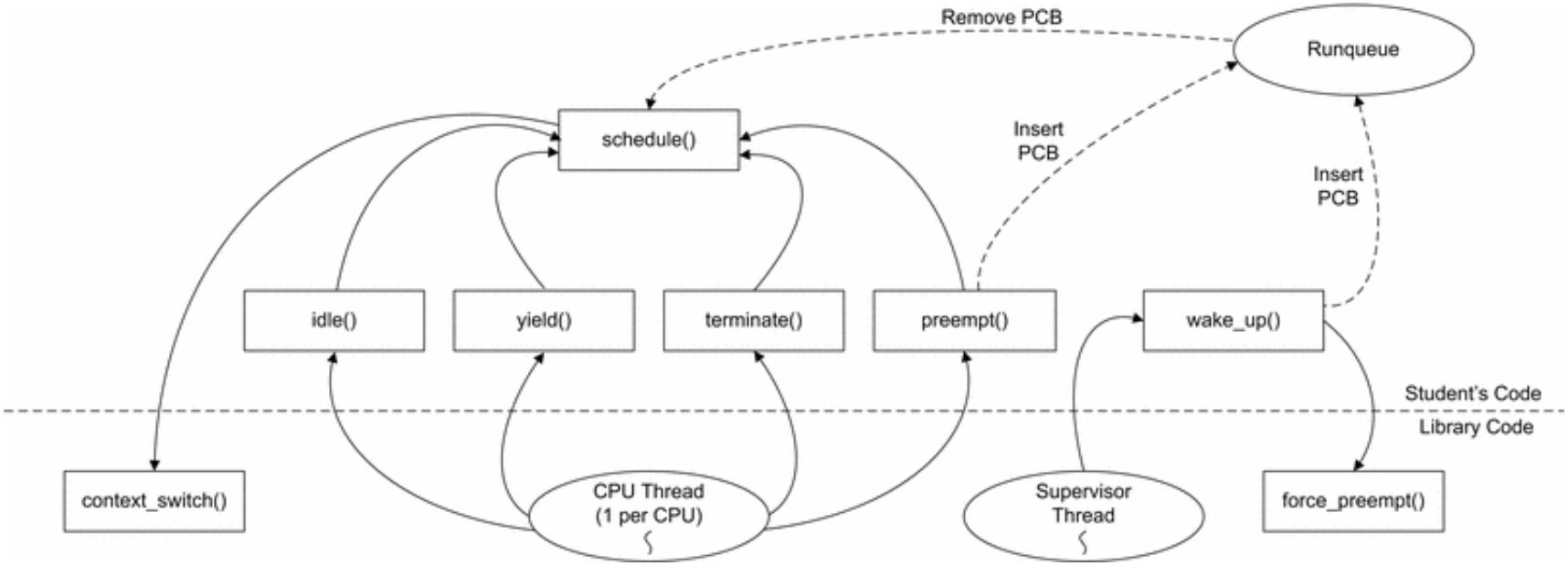

Figure 2: Simulator Function Calls

Sample Output

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)         < <

......
```

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.

Test Processes

For this simulation, we will use a series of eight test processes, five CPU-bound and three I/O-bound. For simplicity, we have labelled each starting with a "C" or "I" to indicate CPU-bound or I/O-bound.

PID	Process Name	CPU / I/O-bound	Priority	Start Time
0	Iapache	I/O-bound	8	0.0 s
1	Ibash	I/O-bound	7	1.0 s
2	Imozilla	I/O-bound	7	2.0 s
3	Ccpu	CPU-bound	5	3.0 s
4	Cgcc	CPU-bound	1	4.0 s
5	Cspice	CPU-bound	2	5.0 s
6	Cmysql	CPU-bound	3	6.0 s

7	Csim	CPU-bound	4	7.0 s

For this project, priorities range from 0 to 10, with 10 being the highest priority. Note that the I/O-bound processes have been given higher priorities than the CPU-bound processes.

Problem 1: FCFS Scheduler

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

- Make sure you understand the addReadyProcess and getReadyProcess functions. You will write similar functions (or modify these with an extra parameter to determine what scheduling algorithm to use) for the Round-Robin and Static Priority scheduling algorithms.
- 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 `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.

Once you successfully complete this portion of the project, test your code with ./os-sim 1, and you should see output similar to the following:

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

```
0.7   1  0  0       Iapache      < <
0.8   1  0  0       Iapache      < <
0.9   1  0  0       Iapache      < <
1.0   0  0  1       (IDLE)       < Iapache <
1.1   1  0  1       Ibash        < Iapache <
1.2   1  0  1       Ibash        < Iapache <
1.3   1  0  1       Ibash        < Iapache <
1.4   1  0  1       Ibash        < Iapache <
1.5   1  0  1       Iapache      < Ibash <
1.6   1  0  1       Iapache      < Ibash <
1.7   0  0  2       (IDLE)       < Ibash Iapache <
1.8   0  0  2       (IDLE)       < Ibash Iapache <
1.9   0  0  2       (IDLE)       < Ibash Iapache <
2.0   1  0  1       Ibash        < Iapache <
....
66.9  1  1  0       Ibash        < <
67.0  1  1  0       Ibash        < <
67.1  1  1  0       Ibash        < <
67.2  1  0  0       Imozilla     < <
67.3  1  0  0       Imozilla     < <
67.4  1  0  0       Imozilla     < <
67.5  1  0  0       Imozilla     < <
```

```
# of Context Switches: 97

Total execution time: 67.6 s

Total time spent in READY state: 389.9 s
```

- Be sure to update the state field of the PCB. The simulator 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 FCFS is non-preemptive, use -1 for this parameter to give the process an infinite timeslice.

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?

Problem 2: Round-Robin Scheduler

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.

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.

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?

Problem 3: Static Priority Scheduling

A. Add Static Priority scheduling to your code. Modify `main()` to accept the `-p` parameter to select the Static Priority algorithm. The `-r` and default FCFS scheduler should continue to work.

The scheduler should use the priority specified in the static_priority field of the PCB. This priority is a value from 0 to 10, with 0 being the lowest priority and 10 being the highest priority.

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 lower priority process when a higher priority process needs a CPU.

B. The Shortest-Job First (SJF) scheduling algorithm is proven to have the optimal average waiting time. However, it is only a theoretical algorithm; it cannot be implemented in a typical CPU scheduler, because the scheduler does not have advance knowledge of the length of each CPU burst.

Run each of your three scheduling algorithms (using one CPU), and compare the total waiting times. Which algorithm is the closest approximation of SJF? Why?