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.

- Make sure you answer all questions (highlighted by **blue font**) in the answers.txt file. Spend some time thinking about these!

Overview

In this project you will implement 2 scheduling algorithms (RR and SP) in a simulator of a simple multiprocessor operating system. You will do this in C using pthreads to get a bit more practice with synchronization in a real context, as well as improving your understanding of how scheduling works in a somewhat realistic OS. The provided framework for the multithreaded OS simulator is technically complete, but currently uses an inefficient non-preemptive FIFO scheduler. Your task is to implement two better 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, especially the comments!

- 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 the FIFO CPU scheduler implementation and functions to manipulate a basic FIFO ready queue. The comments make clear what you do and do not need to modify.
- student.h - Header file for your code to interface with the OS simulator

Scheduling Algorithms

Your scheduler will be capable of using any of the following three CPU scheduling algorithms:

- *First-Come, First Served (i.e. FIFO)* – This is what's implemented currently. Runnable processes are kept in a first-in, first-out 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.
- *Round-Robin* - Similar to FIFO, except preemptive. Each process is assigned a quantum (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.

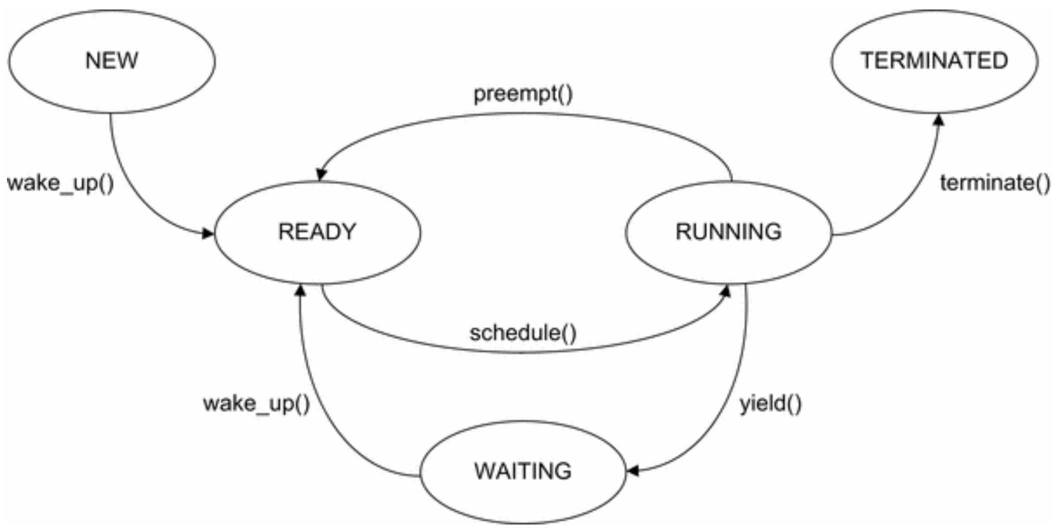
More details and specific steps on how to implement the RR and SP algorithms are specified at the end of this write-up, but first some general info on the simulator code.

<u>Processes</u>
The OS simulator maintains a PCB for each process using a struct of type `pcb_t` (defined in os-sim.h). The PCB contains the pid, name, priority, state, and program counter of the process just as in any real OS. Additionally our simulated OS PCB contains a pointer to the next PCB in whatever queue the current PCB is in (ready or I/O), so that we don't have to create an actual separate linked list for each queue. Your code will manipulate the priority, state, and next fields of the PCB, everything else is used only by the given simulator code.

<u>States</u>
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.

It is important that you update the state field in the PCB anytime your code changes the state of a process, as the simulator uses this field to decide what to do with each process and to collect statistics.



<u>The Ready Queue</u>
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.

<u>Scheduling Processes</u>
`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.

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. Note this is not like yield() in user-level threads where the process goes back onto the ready queue, this yield() specifically means the process is making an I/O system call.
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()`. This 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.

yield(), terminate(), and idle() are already implemented for you and do not require any changes. wake_up() is partially implemented but will need to be modified, and preempt() is given only as a stub that you must fill in with the implementation.

The Simulator

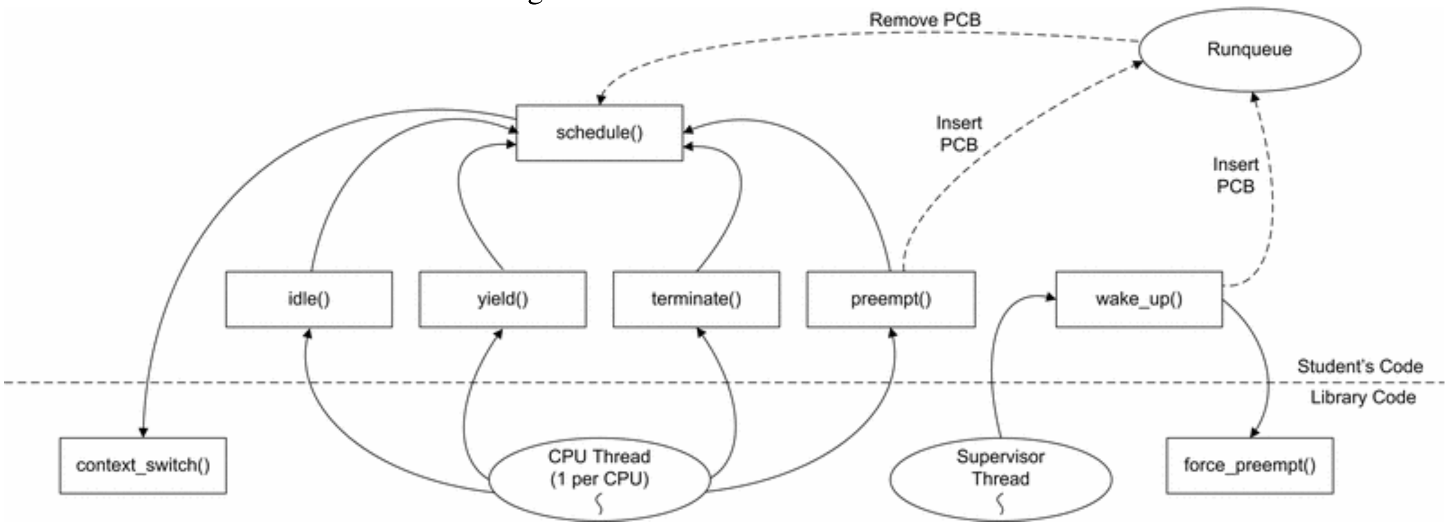
We will use pthreads to simulate an operating system on a multiprocessor computer. We will use one thread per CPU and one thread as a "supervisor" for our simulation. The CPU threads will simulate the currently-running processes on each CPU, and the supervisor thread will print output and dispatch events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

```
running with basic FIFO
starting simulator
Time  Ru Re Wa      CPU 0    CPU 1        < I/O Queue <
===== == == ==      ======== ========     =============
0.0   0  0  0       (IDLE)   (IDLE)       < <
0.1   1  0  0       Iapache  (IDLE)       < <
0.2   1  0  0       Iapache  (IDLE)       < <
0.3   1  0  0       Iapache  (IDLE)       < <
0.4   0  0  1       (IDLE)   (IDLE)       < Iapache <
0.5   0  0  1       (IDLE)   (IDLE)       < Iapache <
0.6   1  0  0       (IDLE)   Iapache      < <
0.7   1  0  0       (IDLE)   Iapache      < <
0.8   1  0  0       (IDLE)   Iapache      < <
0.9   1  0  0       (IDLE)   Iapache      < <
1.0   0  0  1       (IDLE)   (IDLE)       < Iapache <
1.1   1  0  1       Ibash    (IDLE)       < Iapache <
1.2   1  0  1       Ibash    (IDLE)       < Iapache <
1.3   1  0  1       Ibash    (IDLE)       < Iapache <
1.4   1  0  1       Ibash    (IDLE)       < Iapache <
1.5   1  0  1       (IDLE)   Iapache      < Ibash <
1.6   1  0  1       (IDLE)   Iapache      < Ibash <
1.7   0  0  2       (IDLE)   (IDLE)       < Ibash Iapache <
1.8   0  0  2       (IDLE)   (IDLE)       < Ibash Iapache <
1.9   0  0  2       (IDLE)   (IDLE)       < Ibash Iapache <
2.0   1  0  1       Ibash    (IDLE)       < Iapache <
2.1   2  0  1       Ibash    Imozilla     < Iapache <
2.2   2  0  1       Ibash    Imozilla     < Iapache <
2.3   0  0  3       (IDLE)   (IDLE)       < Iapache Ibash Imozilla <
2.4   0  0  3       (IDLE)   (IDLE)       < Iapache Ibash Imozilla <
2.5   1  0  2       Iapache  (IDLE)       < Ibash Imozilla <
2.6   1  0  2       Iapache  (IDLE)       < Ibash Imozilla <
2.7   1  0  2       Iapache  (IDLE)       < Ibash Imozilla <
2.8   0  0  3       (IDLE)   (IDLE)       < Ibash Imozilla Iapache <
2.9   0  0  3       (IDLE)   (IDLE)       < Ibash Imozilla Iapache <
3.0   0  0  3       (IDLE)   (IDLE)       < Ibash Imozilla Iapache <
3.1   1  0  3       (IDLE)   Ccpu         < Ibash Imozilla Iapache <
3.2   2  0  2       Ibash    Ccpu         < Imozilla Iapache <
3.3   2  0  2       Ibash    Ccpu         < Imozilla Iapache <
3.4   1  0  3       (IDLE)   Ccpu         < Imozilla Iapache Ibash <
3.5   1  0  3       (IDLE)   Ccpu         < Imozilla Iapache Ibash <
3.6   1  0  3       (IDLE)   Ccpu         < Imozilla Iapache Ibash <
…
# of Context Switches: 113
Total execution time: 35.8 s
Total time spent in READY state: 83.1 s
```

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.

Test Processes
I have provided eight test processes, five CPU-bound and three I/O-bound. For simplicity, each is labeled 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

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.

FIFO Scheduler

- Look at the code for the FIFO scheduler that is already implemented in student.c Especially make sure you understand the addReadyProcess and getReadyProcess functions. You will modify these for the Round-Robin and Static Priority scheduling algorithms. Note that there is a global variable set in main to tell you which scheduling algorithm you are using. Check out the comments about that variable and make sure to use it correctly as an enum.
- Four of the five entry points into the scheduler `(idle(), yield(), terminate(), preempt())` should cause a new process to be scheduled on the CPU. Note how the already implemented functions call schedule() then context_switch(). When these four functions return, the library automatically simulates the process selected by `context_switch()`.
- `context_switch()` takes a timeslice parameter, which is used for preemptive scheduling algorithms. Since FIFO is non-preemptive, this parameter is currently set to -1 to give the process an infinite timeslice.

Question 1: Run the FIFO 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?

Round Robin Scheduler

Add Round-Robin scheduling functionality to your code. `main()` already handles a command line option, `-r`, to select the Round-Robin scheduling algorithm, and reads another command line argument for the length of the timeslice. For this project, timeslices are measured in tenths of seconds. E.g.

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

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

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

will continue to run the FIFO scheduler.

You don't need to do much to implement Round Robin Scheduling, mainly you have to implement the preempt() function and modify schedule() to use the correct timeslice. 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()` function if the process executes on the CPU for the length of the timeslice without terminating or yielding for I/O.

Question 2: 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?

Static Priority Scheduler

Add Static Priority scheduling to your code. `main()` already handles the command line argument `-p` to select the Static Priority algorithm.

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. You will not change this priority, only use the given value for each process (hence "static").

The `force_preempt()` function provided in os-sim.c preempts a running process before its timeslice expires. Your `wake_up()` function should make use of this function to preempt a lower priority process when a higher priority process needs a CPU. You can modify the getReadyProcess() function to behave differently when using the static priority scheduling algorithm, or write a different getReadyProcess() function for this case. In general though when a process wakes up you will have to check if all processors are being used and the newly woken process has a higher priority than at least one of the currently running processes, in which case you should preempt the lowest priority process currently running and run the newly woken process instead.

Question 3: 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?