CSCI 340 - Operating Systems

Assignment 5 Total Points 100

Objectives

In this assignment you will develop a C program that uses a locking mechanisms (e.g. semaphore or mutex) to solve a concurrency problem. This assignment will allow you to gain experience in the following areas:

- **Threads**: This includes *creating* threads and *joining* threads using the *pthreads* library.
- **Locking mechanism**: You will use a semaphore or mutex locking mechanism in your solution that is provided in the *pthreads* library. For your solution, you will have to make decisions regarding:
  - how many locking variables are needed
  - the initial value of each lock
  - where to call lock/sem_wait and unlock/sem_post for each lock
- **Deadlock** and **Starvation**: As you develop, you will likely encounter deadlock. You will need to determine why you got deadlock, and determine a solution. Though starvation is possible, there should be many example command-line inputs to your program that don't exhibit starvation. In short, if you are ever getting deadlock, or often/always getting starvation, then your solution is wrong.
- **Parallel Debugging**: You will have to debug your code, which is parallel. This is challenging, and using a debugger is often not helpful. Using well-placed *printf()* statements in your code is good way to debug.
- **Non-determinism**: It is normal for your program to exhibit *non-determinism* (ie. The output will likely be different each time you run it, even with the same input parameters). The reason for this is *timing* (of the threads, of the sleep function, of system calls, etc). This is normal. **However**, the output from your code solution should always satisfy the concurrency constraints (listed in the Description section).
- **Simulation Principles**: This assignment is really a simulation. There are are several useful simulation routines provided that you will be using including:
  - sleeping (delaying) for a fraction of a second
  - seeding and generating random numbers *within a multithreaded* program
  - generating a *random integer within a range*

Description

You will be developing the simulator shown in Figure 1. More specifically, the simulator includes four software components that simulate how a real operating system (OS) may: add a job to a priority queue (PQ), schedule jobs in the PQ, remove a job from the PQ, and run a job. A basic description of each software component is provided below.

- **Forker**: A *thread* that simulates the creation of a new job produced by a *fork()* system call.
- **Priority Queue (PQ)**: Is a min-heap data structure implemented using a singly linked list. The value used to create the min-heap will be the shortest remaining time (SRT) priority metric. See table 9.3 on page 405 in the course textbook. The PQ has a maximum capacity (i.e. is bounded).

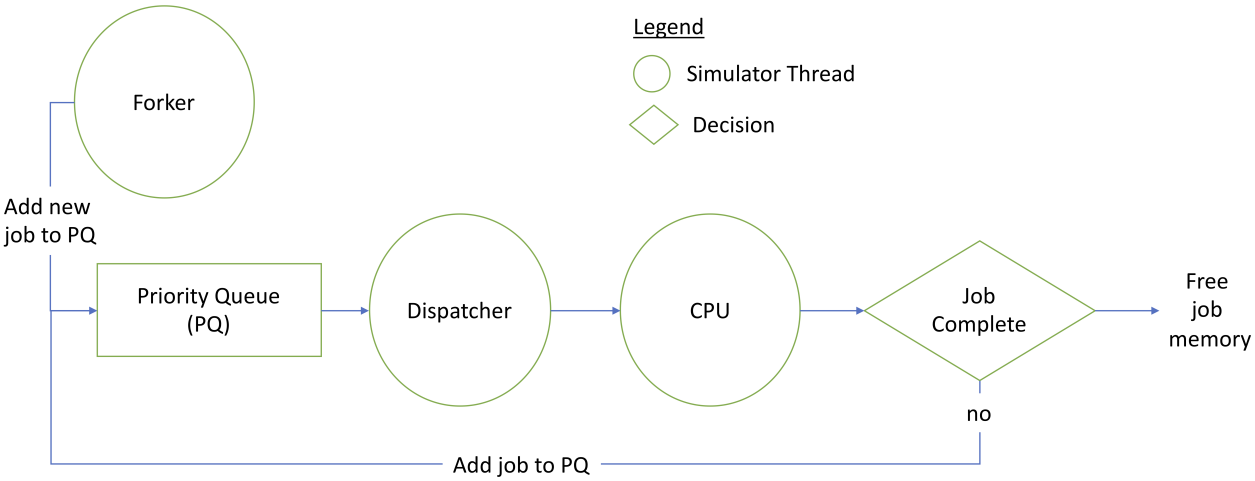

Figure 1: Simulator System Figure

- **Dispatcher**: A *thread* that runs a scheduler (min-heap algorithm) to determine the next job removed from the PQ and then given to the CPU for execution.
- **Central Processing Unit (CPU)**: A *thread* the simulates the CPU. In this simulation, the CPU will only run for one time quantum (or just one loop in a for- or while-loop) and then make a decision to add the job back to the PQ or free the job memory (i.e. job is complete).

In this simulation, there is just one forker, dispatcher, and CPU thread. Here are the concurrency constraints:

1. Forker or CPU cannot add a job to the PQ if it is full.
2. The Dispatcher cannot remove a job from the PQ if it is empty.
3. Only the Forker or CPU (only one thread) may add a job to the PQ at any given time.
4. The dispatcher cannot use the PQ if the forker or dispatcher is adding a job.

Provided Files

The three files listed below are provided to you.

- **simulator.h**: Header file that defines the job_t *struct*, global variables, and constants used in this assignment along with the eight function prototypes listed below.
  - void cpu();
  - void dispatcher();
  - void forker();
  - void scheduler();
  - void initialize()
  - void nsleep();
  - job_t* get_list_element(int index_position);
  - void print_pq();

- **simulator.c**: The file containing the implementation of the functions listed in *simulator.h*. Having a different file for the implementation separates interface (the include file) from the implementation (the .c file).
- **hw5.c**: Source code file that runs the simulation No change should be made to this file.

The *simulator.c* and *simulator.h* files have many comments that provide guidance. Please read the comments carefully!

Todo

In the *simulator.c* file, you must complete the following function implementations that achieve the concurrency constraints defined in the Description section.

```
void initialize()
void scheduler()
void forker()
void cpu()
void dispatcher()
job_t* get_list_element( int index_position )
```

In the *simulator.h* file, you must define all the locking variables (semaphore or mutex, your choice) to achieve the concurrency constraints defined in the Description section. **Note**: This also includes adding the required locking mechanism library header files.

For each listed above, numerous **TODO** comments are provided in the simulator.c and .h files to guide you in this assignment. Read them carefully!

Collaboration and Plagiarism

This is an **individual assignment**, i.e. **no collaboration is permitted**. Plagiarism will not be tolerated. Submitted solutions that are very similar (determined by the instructor) will be given a grade of zero. Do your own work, and everything will be OK.

Submission

Create a compressed tarball, i.e. *tar.gz*, that contains only the completed *simulator.c* and *simulator.h* files. The name of the compressed tarball must be your last name in lower case. For example, *ritchie.tar.gz* would be correct if the original co-developer of UNIX (Dennis Ritchie) submitted the assignment. Only assignments submitted in the correct format will be accepted (no exceptions). Submit the compressed tarball to the appropriate Dropbox on OAKS by the due date. You may resubmit the compressed tarball as many times as you like, only the latest submission will be graded.

To be fair to everyone, late assignments will not be accepted. Exceptions will only be made for extenuating circumstances, i.e. death in the family, health related problems, etc. You will be given **week** to complete this assignment. Poor time management is not an excuse.

Do not email your assignment after the due date. It will not be accepted. If you need help, stop by during office hours to discuss the assignment. I am always happy to listen to your approach and make suggestions. However, I cannot tell you how to code the solution. In addition, code debugging is your job. You should use print statements to help understand why your solution is not working correctly.

Grading Rubric

For this assignment the grading rubric is provided in the table shown below.

reasonable number and initialization of locking variables	5 points
dispatcher() function implementation	15 points
cpu() implementation	15 points
forker() implementation	15 points
scheduler() implementation	10 points
solution compiles and runs	10 points
pass three independent test cases (10 points each)	30 points

NOTE: Few points, if any, will be given to a program which either does not compile or generates a runtime error. Some examples of a runtime error are *segmentation fault*, *divide by zero*, *memory fault*.

Additional Guidance

For details about the heap data structure and the min-heap algorithm, see the *Heap.pdf* document attached to the Dropbox.