CS/SE 4348 Operating Systems
Project 3: Concurrent Threads
Due date April 5th, 2020

Read the project description carefully several times.This project can either be done individually or with a partner. Do not share your work with other groups. Compile the code in any of the cs* machines. However, do not execute in the cs* machines. Use any of the {net01,…, net45} machines for program execution. The number of threads you will create will be large. You do not want to slow down the cs* machines by executing in them. If you are caught executing your program in any of the cs* machines, you will be given a ticket. For every ticket you receive, we will deduct 5 points from your final score.

Seeking Tutor Problem

The computer science department runs a mentoring center (csmc) to help undergraduate students with their programming assignments. The lab has a coordinator and several tutors to assist the students. The waiting area of the center has several chairs. Initially, all the chairs are empty. The coordinator is waiting for the students to arrive. The tutors are either waiting for the coordinator to notify that there are students waiting or they are busy tutoring. The tutoring area is separate from the waiting area.

A student while programming for his project, decides to go to csmc to get help from a tutor. After arriving at the center, the student sits in an empty chair in the waiting area and waits to be called for tutoring. If no chairs are available, the student will go back to programming and come back to the center later. Once a student arrives, the coordinator queues the student based on the student's priority (details on the priority discussed below), and then the coordinator notifies an idle tutor. A tutor, once woken up, finds the student with the highest priority and begins tutoring. A tutor after helping a student, waits for the next student. A student, after receiving help from a tutor goes back to programming.

The priority of a student is based on the number of times the student has visited the tutoring center. A student visiting the center for the first time gets the highest priority. In general, a student visiting for the i^{th} time has a priority higher than the priority of the student visiting for the k^{th} time for any $k > i$. If two students have the same priority, then the student who came first has a higher priority.

Using POSIX threads, mutex locks, and semaphores implement a solution that synchronizes the activities of the coordinator, tutors, and the students. Some hints to implement this project are provided later in the description.

The total number of students, the number of tutors, the number of chairs, and the number of times a student seeks a tutor's help are passed as command line arguments as shown below (csmc is the name of the executable):

```
csmc #students #tutors #chairs #help
csmc 10 3 4 5
csmc 2000 10 20 4
```

Once a student thread takes trequired number of help from the tutors, it should terminate. Once all the student threads are terminated, the tutor threads, the coordinator thread, and the main program should be terminated.
Your program should work for any number of students, tutors, chairs and help sought. Allocate memory for data structures dynamically based on the input parameter(s).

Output

Your program must output the following at appropriate times.

Output of a student thread (*x* and *y* are ids):

```
St: Student x takes a seat. Empty chairs = <# of empty chairs>.
St: Student x found no empty chair. Will try again later
St: Student x received help from Tutor y.
```

Output of coordinator threads (*x* is the id, and *p* is the priority):

```
Co: Student x with priority p in the queue. Waiting students now
= <# students waiting>. Total requests = <total # requests
(notifications sent) by students for tutoring so far>
```

Output of a tutor thread after tutoring a student (*x* and *y* are ids):

```
Tu: Student x tutored by Tutor y. Students tutored now = <#
students receiving help now>. Total sessions tutored = <total
number of tutoring sessions completed so far by all the tutors>
```

Your program should not output anything else. You may use several printf statements for debugging. Disable them before you submit the code. You will lose one point for every line of unspecified output. If you are testing your program for a large number of threads, redirect the output to a file. It will save time for you. Also, print all error messages on stderr. This will avoid the error message redirection along with the output.

Implementation Hints

Using Pthreads, begin by creating n students and m tutors as separate threads. (n and m are arguments to the program.) The coordinator will run as a separate thread. Student threads will alternate between programming for a period of time and seeking help from the tutor. If the tutor is available, they will obtain help. Otherwise, they will either sit in a chair in the waiting area or, if no chairs are available, will resume programming and seek help at a later time.

When a student arrives and finds an empty chair, the student must notify the coordinator using a semaphore. The coordinator then has to queue the student according to the student's priority. How does the coordinator know which student has arrived? How does the tutor find the student with the highest priority? (Clue: You need two shared data structures here. Also, you just need a simple data structure for implementing priority.) When a tutor is free (either initially or after helping a student), the tutor must wait for the coordinator to notify of a waiting student (use another semaphore). A tutor should then wake up the student with highest priority. A tutor cannot wake up any student.

Simulate the programming part of a student thread by sleeping for a random amount of time up to 2 ms. For the tutoring part, make both the student and the tutor thread sleep for 0.2 ms.

For details on how to use pthreads, synchronization primitives mutex and semaphores see man pages. For a more detailed tutorial on Pthreads and Semaphores, see https://computing.llnl.gov/tutorials/pthreads/#Thread

You may also want to look at the solution for sleeping barber problem. It will give you some clues about how to solve the problem this project poses.

<u>Grading Policy</u>

Name your program file as csmc.c and submit it in the directory /CS4348-xv6/xxxyyyyyy/P3. Do not submit it in elearning.

Correctness: 80%.

Style: 20%. Source code should be well structured with adequate comments clearly describing the different parts and functionalities implemented.

The TA will compile the code in any of the cs* machine and execute in net* machines. If your code does not compile in any of the cs* machines you will get 0 points. You don't need to check whether your program compiles in all the cs* machines. If it compiles in one of the cs* machines, it will compile in all. Add a note at the top of the source code in case you are using any special flags for compilers/linkers.

You have to demonstrate your work to the TA. If you are not able to explain how your code works, then you will not get any points even though your code may work.

Concurrent programming is fun. Enjoy!!!