Group Project 2: Application for Multithreaded Synchronization

CECS 326 – Operating Systems

1. Summary

This project follows the topic of the first project and will ask you to design a real scenario/application with multithreaded synchronization. You will use the threads programming interface, POSIX Threads (Pthreads). You should implement this in Linux, which supports Pthreads as part of the GNU C library.

You should submit the required deliverable materials on BeachBoard by **11:55pm, October 18st (Sunday), 2020.**

2. Description

You have been hired by the Department of CECS at CSULB to write code to help synchronize a professor and his/her students during office hours. The professor, of course, wants to take a nap if no students are around to ask questions; if there are students who want to ask questions, they must synchronize with each other and with the professor so that

(i) no more than a certain number of students can be in the office at the same time because the **office has limited capacity**,
(ii) only one person is speaking at a time,
(iii) each student question is answered by the professor,
(iv) no student asks another question before the professor is done answering the previous one
(v) once a student finishes asking all his/her questions, he/she must leave the office to make room for other students waiting outside.

You are to provide the following functions:

- ***Professor()***. This functions starts a thread that runs a loop calling ***AnswerStart()*** and AnswerDone(). See below for the specification of these two functions. ***AnswerStart()*** blocks when there are no students around.
- ***Student(int id).*** This function creates a thread that represents a new student with identifier id that asks the professor one or more questions (the identifier given to your function can be expected to be greater or equal to zero and the first student's id is zero). First, each student needs to enter the professor's office by calling ***EnterOffice().*** If the office is already full, the student must wait. After a
- student enters the office, he/she loops running the code ***QuestionStart()*** and ***QuestionDone()*** for the number of questions that he/she wants to ask. The number of questions is determined by calculating (student identifier modulo 4 plus 1). That is, each student can ask between 1 and 4 questions, depending on the id. For example, a student with id 2 asks 3 questions, a student with id 11 asks 4 questions and a student with id 4 asks a single question. Once the student has got the answer for all his/her questions, he/she

must call ***LeaveOffice().*** As a result, another student waiting on ***EnterOffice()*** may be able to proceed.

- ***AnswerStart()***. The professor starts to answer a question of a student. Print ... Professor starts to answer question for student x.
- ***AnswerDone()***. The professor is done answering a question of a student. Print ... Professor is done with answer for student x.
- ***EnterOffice()***. It is the student's turn to enter the professor's office to ask questions. Print ... Student x shows up in the office.
- ***LeaveOffice()***. The student has no more questions to ask, so he/she leaves the professor's office. Print ... Student x leaves the office.
- ***QuestionStart()***. It is the turn of the student to ask his/her next question. Print ... Student x asks a question. Wait to print out the message until it is really that student's turn.
- ***QuestionDone()***. The student is satisfied with the answer to his most recent question. Print ... Student x is satisfied.
- Since professors consider it rude for a student not to wait for an answer, ***QuestionDone()*** should not print anything until the professor has finished answering the question.

A student can ask only one question each time. I.e., a student should not expect to ask all his/her questions in a contiguous batch. In other words, once a student gets the answer to one of his/her questions, he/she may have to wait for the next turn if another student starts to ask a question before he/she does.

In the above list, x is a placeholder for the student identifier.

Your program must **accept one command line parameter** that represents the number of students coming to the professor's office, and a second command line parameter that represents the capacity of the professor's office (i.e., how many students can be in the office at the same time). For simplicity, you can assume that the Student threads are created at the ascending order of their identifiers.

Your program must validate the command line parameters to make sure that they are numbers, not arbitrary garbage.

Your program must be able to run properly with any reasonable number of students (e.g., 100) and room capacity (e.g., 8).

One **sample output** of your program is (assuming 3 students and a room capacity of 2):

```
Student 0 shows up in the office.
Student 1 shows up in the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 0 asks a question.
Professor starts to answer question for student 0.
Professor is done with answer for student 0.
Student 0 is satisfied.
Student 0 leaves the office.
Student 2 shows up in the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 1 leaves the office.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 2 leaves the office.
```

3: The Required Deliverable Materials

(1) A README file, which describes how we can compile and run your code.
(2) Your source code, must include a Makefile and be submitted in the required format.
(3) Your report, which discusses the design of your Office Hour program and how Pthread synchronization is used in your program.

3. Submission Requirements

You need to strictly follow the instructions listed below:

1) This is a **group project**, please submit a .zip/.rar file that contains all files, only one submission from one group.

2) The submission should include your **source code** and **project report**. Do not submit your binary code. Project report should contain your groupmates name and ID.

3) Your code must **be able to compile**; otherwise, you will receive a grade of zero.

4) Your code should not produce anything else other than the required information in the output file.

5) Your code must validate command line parameters to make sure that only numbers can be accepted.

6) If you code is **partially completed**, please explain the details in the report what has been completed and the status of the missing parts, we will grade it based on the entire performance.

7) Provide **sufficient comments** in your code to help the TA understand your code. This is important for you to get at least partial credit in case your submitted code does not work properly.

Grading criteria:

Details	Points
Submission follows the right formats	5 pts
Have a README file shows how to compile and test your submission	5 pts
Submitted code has proper comments to show the design	10 pts
Have a **report** (pdf or word) file explains the details of your entire design	25 pts
Report contains clearly individual contributions of your group mates	10 pts
Code can be compiled and shows correct outputs	45 pts

4. Policies

1) Late submissions will be graded based on our policy discussed in the course syllabus. 2) Code-level discussion is **prohibited**. We will use anti-plagiarism tools to detect violations of this policy.

5. Resources

The Pthreads tutorials at https://computing.llnl.gov/tutorials/pthreads and

http://pages.cs.wisc.edu/~travitch/pthreads_primer.html are good references to learn Pthreads programming.

6. References

[1] POSIX Threads Programming: https://computing.llnl.gov/tutorials/pthreads/

[2] Pthreads Primer: http://pages.cs.wisc.edu/~travitch/pthreads_primer.html

[3] POSIX thread (pthread) libraries:
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html