Multithreaded Programming and Synchronization

COP 5614 — Operating Systems

1. Summary

This project is regarding several important topics on process management. But instead of developing it in kernel, we will do it in user space using a widely-used threads programming interface, POSIX Theads (Pthreads). You should implement this in Linux, which supports Pthreads as part of the GNU C library.

2. Description

In this assignment, you will be working with the "threads"' subsystem of Linux. This is the part of Linux that supports multiple concurrent activities within the kernel. In the exercises below, you will write a simple program that creates multiple threads, you will demonstrate the problems that arise when multiple threads perform unsynchronized access to shared data, and you will rectify these problems by introducing synchronization (in the form of Pthreads mutex) into the code. Then, you will simulate one real world synchronization problem by using the synchronization primitives supplied by Pthreads.

Part 1: Simple Multi-thread Programming

The purpose of this exercise is for you to get some experience using the threads primitives provided by Pthreads [1], and to demonstrate what happens if concurrently executing threads access shared variables without proper synchronization. Then you will use the mutex synchronization primitives in Pthreads to achieve proper synchronization.

Step 1.1: Simple Multi-thread Programming without Synchronization - Weight: 20%
First, you need to write a program using the Pthread library that forks a number of threads each executes the loop in the SimpleThread function below. The number of threads is a command line parameter of your program. All the threads modify a shared variable SharedVariable and display its value within and after the loop.

```
int SharedVariable = 0;

void SimpleThread(int which) {
    int num, val;

    for(num = 0; num < 20; num++) {
        if (random() > RAND_MAX / 2)
            usleep(500);

        val = SharedVariable;
        printf("*** thread %d sees value %d\n", which, val);
        SharedVariable = val + 1;
    }
    val = SharedVariable;
    printf("Thread %d sees final value %d\n", which, val);
}
```

Your program must validate the command line parameter to make sure that it is a number, not arbitrary garbage.

Your program must be able to run properly with any reasonable number of threads (e.g., 200).

Try your program with the command line parameter set to 1, 2, 3, 4, and 5. Analyze and explain the results. Put your explanation in your project report.

Step 1.2: Simple Threads Programming with Proper Synchronization- Weight: 30%
Modify your program by introducing pthread mutex variables, so that accesses to the shared variable are properly synchronized. Try your synchronized version with the command line parameter set to 1, 2, 3, 4, and 5. Accesses to the shared variables are properly synchronized if (i) **each iteration of the loop in SimpleThread() increments the variable by exactly one** and (ii) **each thread sees the same final value**. It is necessary to use a pthread barrier [2] in order to allow all threads to wait for the last to exit the loop.

You must surround *all* of your synchronization-related changes with preprocessor commands, so that we can easily compile and get the version of your program developed in Step 1.1. E.g.,

```
for(num = 0; num < 20; num++) {
#ifdef PTHREAD _SYNC
      /* put your synchronization-related code here */
#endif
    val = SharedVariable;
    printf("*** thread %d sees value %d\n", which, val);
    SharedVariable = val + 1;
    …….
}
```

One acceptable output of your program is (assuming 4 threads):
*** thread 0 sees value 0
*** thread 0 sees value 1
*** thread 0 sees value 2
*** thread 0 sees value 3
*** thread 0 sees value 4
*** thread 1 sees value 5
*** thread 1 sees value 6
*** thread 1 sees value 7
*** thread 1 sees value 8
*** thread 1 sees value 9
*** thread 2 sees value 10
*** thread 2 sees value 11
*** thread 2 sees value 12
*** thread 3 sees value 13
*** thread 3 sees value 14
*** thread 3 sees value 15
*** thread 3 sees value 16
*** thread 3 sees value 17
*** thread 2 sees value 18
*** thread 2 sees value 19
……
Thread 0 sees final value 80
Thread 2 sees final value 80

Thread 1 sees final value 80
Thread 3 sees final value 80

Step 1.3: Part 1 deliverables:

(1) A README file, which describes how we can compile and run your code;
(2) Your source code, must include a Makefile;
(3) Your report, which discusses the output of your program without Pthread synchronization and the one with Pthread synchronization, as well as the reason for the difference.

Part 2: Office Hours Multi-Thread Programming - Weight: 50%

You have been hired by the School of Computer Science at FIU 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, and (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 enters 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 acceptable output of your program is (assuming 3 students and a room capacity of 2):

```
Student 0 enters the office.
Student 1 enters 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 enters 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.
```

Part 2 deliverables:

(1) A README file, which describes how we can compile and run your code;
(2) Your source code, must include a Makefile;
(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) The submission should include deliverables for both Part 1 and Part 2. Submit a .zip file that contains all files; put Part 1 and Part 2 in separate directories.
2) The submission should include only your source code and report. Do not submit your binary code.
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, also explain in the report what has been completed and the status of the missing parts.
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.
8) Clearly state **your group members** and **who does what** in your report.

4. Policies

1) Late submissions will not be graded.
2) You must work in a group of four people on this exercise. We encourage high-level discussions among the groups to help each other understand the concepts and principles. However, 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.

If you are new to C programming, “The C Cheat Sheet” at http://claymore.engineer.gvsu.edu/~steriana/226/C.CheatSheet.pdf may be a good starting point.

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
[4] http://claymore.engineer.gvsu.edu/~steriana/226/C.CheatSheet.pdf