Lab Exercise 2: Multithreaded Programming and Synchronization

COP 4610 — Operating Systems Principles

Demo in Lab sessions: U01 - Tue Feb 21; U02: Fri Feb 24

1. Summary

The second lab exercise is regarding several important topics on process management. You will do a little bit of work inside the kernel to be familiar with kernel data structures related to processes and threads. For the remaining part, you will do it in user space using a widely-used threads programming interface, POSIX Theads (Pthreads). You should implement this exercise on the VM you used in Lab 1, which supports Pthreads as part of the GNU C library.

You should submit the required deliverable materials on Moodle by **11:55pm, February 24, 2017 (Friday)**.

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. You will also add a new system call, which allows you to inspect process and thread information from the kernel. 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 thredID) {
    int num, val;

    for(num = 0; num < 20; num++) {
        if (random() > RAND_MAX / 2)
            usleep(10);

        val = SharedVariable;
        printf("*** thread %d sees value %d\n", thredID, val);
```

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

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

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: 20%
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
            /* include 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: Inspect Processes and the Threads in Your Program in the Kernel- Weight: 10%
Create a new system call that can dump information about all processes in the system and the threads in your multi-threaded program. For each process or thread, you need to dump its command line, state (running, waiting, etc), process id, and its parent's process id. The dumping can be done using **printk**. You can use the same method as in Lab 1 to create this new system call. The name of the system call should be sys_lastnames_of_group_members_in_ascending_order.

Modify your program so that it calls your new system call after all threads have been created but before they are terminated (while all treads` are alive). Use **dmesg** to verify that information about all threads of your program can be dumped inside your system call. For example, if your program forks 3 threads, **dmesg** output should show all 3 of them.

Hint: use the macros `for_each_process` and `while_each_thread` to iterate through the list of process control blocks in Linux. More information can be found in [6][7].

Step 1.4: 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) A patch file that represents the changes that you made to the vanilla 2.6.36 kernel to implement the new system call. It is okay to include the other system call that you added in Lab 1;
(4) 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. In addition, the output of **dmesg** that shows the process and thread information.

Part 2: News Conference Multi-Thread Programming - Weight: 40%
Suppose that you have been hired by the government to write code to help synchronize a speaker and the media reporters during a news conference. The speaker, of course, wants to take a break if no reporters are around to ask questions; if there are reporters who want to ask questions, they must synchronize with each other and with the speaker so that (i) no more than a certain number of reporters can be in the conference room at the same time because **the conference room has limited capacity**, (ii) only one person is speaking at a time, (iii) each reporter's question is answered by the speaker, (iv) no reporter asks another question before the speaker is done answering the previous one, and (v) once a reporter finishes asking all his/her questions, he/she must leave the conference room to make room for other reporters waiting outside.

You are to provide the following functions:

- *Speaker()*. 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 reporters around.
- *Reporter(int id)*. This function creates a thread that represents a new reporter with identifier id that asks the Speaker one or more questions (the identifier given to your function can be expected to be greater or equal to zero and the first reporter's id is zero). First, each reporter needs to enter the conference room by calling EnterConferenceRoom(). If the conference room is already full, the reporter

must wait. After a reporter enters the conference room, 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 (reporter identifier modulo 4 plus 2). That is, each reporter can ask between 2 and 5 questions, depending on the id. For example, a reporter with id 2 asks 4 questions, a reporter with id 11 asks 5 questions and a reporter with id 4 asks 2 questions. Once the reporter has got the answer for all his/her questions, he/she must call LeaveConferenceRoom(). As a result, another reporter waiting on EnterConferenceRoom() may be able to proceed.

- *AnswerStart()*. The Speaker starts to answer a question of a reporter. Print ...
  Speaker starts to answer question for reporter x.
- *AnswerDone()*. The Speaker is done answering a question of a reporter. Print ...
  Speaker is done with answer for reporter x.
- *EnterConferenceRoom()*. It is the reporter's turn to enter the conference room to ask questions. Print …
  Reporter x enters the conference room.
- LeaveConferenceRoom(). The reporter has no more questions to ask, so he/she leaves the conference room. Print …
  Reporter x leaves the conference room.
- *QuestionStart()*. It is the turn of the reporter to ask his/her next question. Print ...
  Reporter x asks a question.
  Wait to print out the message until it is really that reporter's turn.
- *QuestionDone()*. The reporter is satisfied with the answer to his most recent question. Print ...
  Reporter x is satisfied.

Since it is generally considered rude for a reporter not to wait for an answer, *QuestionDone()* should not print anything until the Speaker has finished answering the question.

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

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

Your program must accept one command line parameter that represents the number of reporters coming to the news conference, and a second command line parameter that represents the capacity of the conference room (i.e., how many reporters can be in the conference room at the same time). For simplicity, you can assume that the Reporter 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 reporters (e.g., 100) and room capacity (e.g., 8).

One acceptable output of your program is (assuming 4 reporters and a room capacity of 3):

```
Reporter 0 enters the conference room.
Reporter 1 enters the conference room.
Reporter 1 asks a question.
Speaker starts to answer question for reporter 1.
Speaker is done with answer for reporter 1.
Reporter 1 is satisfied.
Reporter 0 asks a question.
Speaker starts to answer question for reporter 0.
Speaker is done with answer for reporter 0.
Reporter 0 is satisfied.
Reporter 2 enters the conference room.
Reporter 1 asks a question.
Speaker starts to answer question for reporter 1.
Speaker is done with answer for reporter 1.
Reporter 1 is satisfied.
Reporter 2 asks a question.
Speaker starts to answer question for reporter 2.
Speaker is done with answer for reporter 2.
Reporter 2 is satisfied.
Reporter 0 asks a question.
Speaker starts to answer question for reporter 0.
Speaker is done with answer for reporter 0.
Reporter 0 is satisfied.
Reporter 0 leaves the conference room.
Reporter 1 asks a question.
Speaker starts to answer question for reporter 2.
Speaker is done with answer for reporter 2.
Reporter 1 is satisfied.
Reporter 3 enters the conference room.
Reporter 2 asks a question.
Speaker starts to answer question for reporter 2.
Speaker is done with answer for reporter 2.
Reporter 2 is satisfied.
Reporter 1 leaves the conference room.
……
```

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 (10%), which discusses the design of your News Conference 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 be graded based on our policy discussed in the course syllabus.
2) You must work in a group of three 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 and the Makefile tutorial in Moodle 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
[5] A simple Makefile tutorial: Moodle
[6] http://stackoverflow.com/questions/19208487/kernel-module-that-iterates-over-all-tasks-using-depth-first-tree
[7] http://stackoverflow.com/questions/8457890/kernel-how-to-find-all-threads-from-a-processs-task-struct