Lab 2
Pthreads Synchronization

In this lab, you will learn the basics of multi-threaded programming, and synchronizing multiple threads using locks and condition variables. You will use the pthreads thread API in this assignment.

Before you begin

Please familiarize yourself with the pthreads API thoroughly. Many helpful tutorials and sample programs are available online. Practice writing simple programs with multiple threads, using locks and condition variables for synchronization across threads. Some example programs for you to write are:

• Write a program that has a counter as a global variable. Spawn 10 threads in the program, and let each thread increment the counter 1000 times in a loop. Print the final value of the counter after all the threads finish—the expected value of the counter is 10000. Run this program first without using locking across threads, and observe the incorrect updation of the counter due to race conditions (the final value will be slightly less than 10000). Next, use locks when accessing the shared counter and verify that the counter is now updated correctly.

• Write a program with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order. Now, add synchronization to your code such that the numbers are printed in the order 1, 2, ..., N , 1, 2, ..., N , and so on. You may want to start with N = 2 and then move on to larger values of N .

Semaphores using pthreads

In this part of the lab, you will implement the synchronization functionality of semaphores using pthreads mutexes and condition variables. Let's call these new userspace semaphores that you implement as zemaphores, to avoid confusing them with semaphores provided by the Linux kernel. You must define your zemaphore structure in the file zemaphore.h, and implement the functions zem init, zem up and zem down that operate on this structure in the file zemaphore.c. The semantics of these zemaphore functions are similar to those of the semaphores you have studied in class.

• The function zem init initializes the specified zemaphore to the specified value.

• The function zem up increments the counter value of the zemaphore by one, and wakes up any one sleeping thread.

• The function zem down decrements the counter value of the zemaphore by one. If the value is negative, the thread blocks and is context switched out, to be woken up by an up operation on the zemaphore at a later point.

Once you implement your zemaphores, you can use the program test-zem.c to test your implementation. This program spawns two threads, and all three threads (the main thread and the two new threads) share a zemaphore. Before you implement the zemaphore logic, the new threads will print to screen before the main thread. However, after you implement the zemaphore correctly, the main thread will print first, owing to the synchronization enabled by the zemaphore. You must not modify this test program in any way, but only use it to test your zemaphore implementation.

Next, you are given a simple program with three threads in test-toggle.c. In its current form, the three threads will all print to screen in any order. Modify this program in such a way that the print statements of the threads are interleaved in the order thread0, thread1, thread2, thread0, thread1, thread2,... and so on. You must only use your zemaphores to achieve this synchronization between the threads. You must not directly use the mutexes and condition variables of the pthreads library in the file test-toggle.c. The script test-zem.sh will compile and run these test programs for you and can be used for testing.

Submission:

Please submit a zip file ***<Group_number>.zip*** with the following contents

1. Program: you have to submit ***zemaphore.h, zemaphore.c, test-toggle.c***
2. Report: ***<group_number>.<extension>*** (e.g., 1.pdf). Report should be in pdf format.
3. Readme file: readme.txt.