CSCI 5573 Homework Assignment 5 Spring 2022

Due Date: 11:59 pm, April 11 (Mon), 2022

1) Disclaimer

This homework is created based on our textbook, Operating Systems: Three Easy Pieces, written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin.

2) Goal

In this homework, we'll use semaphores to solve some well-known concurrency problems.

3) Requirement

- Download all skeleton of codes from the class Canvas.
- You need to complete all code and test your codes in Linux OS, csegrid.

4) What to submit & how to submit

a) Answer all questions and add print output for each question.
b) Submit all codes you completed.
c) Upload all required materials to the class Canvas.

5) Questions

Each of the following questions provides a code skeleton; your job is to fill in the code to make it work given semaphores. On Linux, you will be using native semaphores; You'll have to first build an implementation (using locks and condition variables, as described in the chapter).

1. The first problem is just to implement and test a solution to the **fork/join problem**, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See **fork-join.c** for details. Add the call sleep(1) to the child to ensure it is working.

2. Let's now generalize this a bit by investigating the **rendezvous problem**. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see **rendezvous.c** for detail.s

3. Now go one step further by implementing a general solution to **barrier synchronization**. Assume there are two points in a sequential piece of code, called $P1$ and $P2$. Putting a **barrier** between $P1$ and $P2$ guarantees that all threads will execute $P1$ before any one thread executes $P2$. Your task: write the code to implement a barrier() function that can be used in this man- ner. It is safe to assume you know *N* (the total number of threads in the running program) and that all *N* threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See **barrier.c** for details.

4. Now let's solve the **reader-writer problem**, also as described in the text. In this first take, don't worry about starvation. See the code in **reader-writer.c** for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?

5. Let's look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See **reader-writer-nostarve.c** for details.

6. Use semaphores to build a **no-starve mutex**, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in **mutex-nostarve.c** for more information.