Cosc 4740 – Lab04

Using Semaphores to avoid race conditions.

1.1 Starting Repo

Accept the GitHub lab here: https://classroom.github.com/a/rXi64A3y
Starting repo will have a standard readme and initial code for you in: thread_semaphore.cpp

1.2 Background

Finite resources need to be managed. Suppose there is only one resource. Processes may ask for any number of this resource and return them when finished.

For example, many commercial software packages provide a limited number of application licenses, representing the number of time that application can be run at the same time. When an application is run the license count decrements, and when it's no longer in use it increments. When all the licenses are in use, a new instance of the application cannot be started until a previous instance is finished.

1.3 Goal

You will be designing and implementing the functionality described above and emulate its behavior.

The program will hold a single finite resource that an application thread will want to use. At the start 10 application threads will be launched, each of which will try to use 1-3 of the resource chosen at random. The application threads will use them for a short time and then return them.

You will be using semaphores and/or mutex locks to fix any race conditions when accessing the resource.

1.4 Task/Code

A stub of the code has been provided in the thread_semaphore.cpp file provided to you in the starting GitHub repo. You need to complete the code and follow any instructions listed in the comments. There will be two parts to complete.

Before making any modifications, read through the code and then compile and run it.

```
g++ thread_semaphore.cpp -pthread
```

Run it several times. Notice that we have a maximum number of 5 available resources; due to the race condition the program can end with more or less resources available than the maximum.

Part 1:

Identify the race conditions of the two functions, increase_count() and decrease_count() and use a mutex lock or binary semaphore to fix it. Makes sure to look at the comments.

Part 2:

Implement two new functions, increase_S() and decrease_S() and use a counting semaphore so that the application threads don't have to "busy wait" for resources. Again, make sure to look at the comments.

Note: When the program finishes it should always report 5 available resources, any more or less, then something isn't right! Also be sure to avoid all deadlock and race conditions.

1.5 Submission

1. Your code pushed to git
2. An updated readme pushed to git
3. Do not include any other files in your repo