CSC 4730 Fall 2023 Project 6

In this project you will build a semaphore from a condition variable and a mutex.

Objectives

The education purposes of this project include:

- Mastery of the `pthreads` API.
- Understanding the nature and construction of a semaphore.
- Understand and perhaps experience, race conditions.

Deadlines

You and your assigned partner have 7 days to complete this assignment plus one grace day. If you have not turned anything in by 11:59 PM on the eighth day, you will win a grade of zero. Therefore, to receive partial credit, hand in something before the project expires!

Overview

Players gather around a circle and pass something between them as quickly as possible. Called, Hot Potato, imagine the item being tossed around is a potato straight out of the fire.

In our game, we will support any number of potatoes simultaneously from 1 up to the number of players.

Environment

You must use WSL on Windows or the Terminal on the Mac.

Prohibitions

You cannot:

- use `rand()` and its cousins.
- use C++ semaphores.
- use pthreads semaphores.

Requirements

We are going to use pthreads threads and synchronization primitives.

You need a mutex for implementing the semaphore and another one which you must use to ensure no thread's output stomps over another thread's output.

Command Line Options

You must support:

`-h` to print help / usage.

`-n` to set the number of players. The value defaults to 4. It cannot be less than 1 or more than 32.

`-p` to set the number of potatoes. The value defaults to 1. It cannot be less than 1 nor more than the number of players.

You must handle each of the above and do appropriate error checking Should any errors be found you must print some reasonable error messages and quit.

Data Structures

You will need, perhaps, C++ `vectors` or `arrays` to hold counters, possession values and pids.

You must maintain a count for each thread that holds the number of times each player (thread) has held the potato.

You must maintain a `bool` that indicates whether or not a given thread possesses the potato at the moment.

These two data structures form the basis of the program's output.

Discussion

The semaphore must be programmed to allow the number of hot potatos specified by the user on the command line to be active at any one time.

Semaphore Functions

Your implementation of semaphores must follow these signatures:

```
void SemInit(Sem & s, int32_t initial_value);
void SemPost(Sem & s);
int32_t SemWait(Sem & s);
```

Output During Set Up

You must print status information prior to releasing the potatos. For example:

```
Number of children: 4
Number of potatos:  1
Child:  0 has started
Child:  2 has started
Child:  1 has started
Child:  3 has started
Hit return to put potatos in play:
```

Notice that execution is paused by waiting for the keyboard until you hit return.

Sleeping

`sleep()` takes an integer number of seconds as its parameter. Sleeping for seconds would make for a very boring project. Instead use `nanosleep()` which takes a pointer to a `timespec` plus a `nullptr`.

The older `usleep` still works but is far less capable than `nanosleep()` in its ability to cooperate with other aspects of Linux and Unix-like OS's.

The "players" must sleep between 1,000 and 50,000 **microseconds** while holding a potato **and** also after releasing the potato. Note the use of a random number. Your go-to old buddy `rand()` cannot be used in this project because it is not **thread safe**. Instead, use the new C++ random number generation tools.

Restated, you must have some randomness in your solution but you cannot use `srand()` or `rand()` or its cousins. Instead, you must use the C++ `random`.

What's so bad about `rand()`

As stated above, `rand()` is not thread-safe. It maintains an internal state which is a **shared resource**. The state is **not protected**. Therefore, use by multiple threads can cause inconsistent results. So what, you say. It's all random anyway isn't it?

No. `rand()` can be used to generate repeatable sequences of seemingly random numbers. Using it in a threaded program introduces race conditions, the results of which cannot be expected to be repeatable.

Infinite Loops

The players and the main loop should be infinite loops. Terminate the application by ˆC.

ˆs / ˆq

Control-s suspends terminal output. Control-q starts it again. This can be used as a "pause" button while your program is running. Use this to count asterisks.

These terminal sequences descend from the earliest of I/O devices.

Building Your Program

Here is a good `makefile` which will build a single executable out of whatever .cpp files are in the directory while also building a dependency chain (so that should any .h or .hpp files be modified, the right files will be rebuilt).

There is also a `clean` target to clean up the directory. The application will be built with `pthread` linked in. On the Mac this isn't needed but it doesn't hurt. On Linux, it is necessary.

Setting Expectations

My implementation, without many comments, but with blank lines and all self-contained is 168 lines. This is not a challenge. Rather it is to set your expectations. If you, for example, find yourself writing double this number of lines, you should ask yourself if you aren't either mistaken or are working too hard.

Output

Your threads are running asynchronously, updating their own values in the counter and possession vectors.

Four times per second, the main thread will print out the values.

Here is an example:

```
*1380 *1322 *1346  1338  1341 *1325  1332  1320 variance: 4.35
```

Notice there are 8 numbers before the word "variance". These numbers indicated the number of times each thread has held a potato in the past.

Notice four numbers are preceded by "*". These are the players who are in possession of a potato *right now*.

NOTE that the numbers of "*" **might be less** than the number of hot potatos but **will NEVER** be more. If you see more "*" than the number of potatos, you have a bad bad bug.

The "variance" is the largest value minus the smallest value divided by the largest value times 100. Over time, as the other numbers grow, the variance will get smaller. If it does not get smaller over time, you have a bug.

Here is a video showing the output of the project as it runs.

What to Hand In

Zip your makefile and all of your .c and / or .cpp source files and any header files into a single zip file containing NO sub-folders. If your zip file contains a subfolder, I will remove 5 points.

Partners

- You must use only the partner I assign you.
- Only 1 person should hand in code. The code should clearly state who the partners are in a comment at the top of the main source file.

- The non-code-submitting partner must submit a text file “partner.txt” that states who the partners are.
- Failure to list partners correctly as described above removes 5 points from your grade.

Software Kills

In your dream job at a video game maker, you are assisting in adding multi-threading support to a previously CPU bound shooter. You goof. A player becomes so upset with an uncalled-for death in a Battle Royale that they hang themselves. The player’s mom finds him. She is devastated and cannot recover. She kills herself three years later but not before winning a lawsuit against your employer that puts them out of business.

Grading

The class will receive a rubric when grading is complete that describes errors and penalties.

Both partners get the same grade without exception.