Lab

Synchronization II

Race Conditions

First, download `race.c` and read the source code. You should then be able to verify that the output of this program should be 2870. Run the program once, then 40 times in a row, then 500 times in a row. A `bash` script can help you do this:[1]

```
for i in {1..40}; do ./a.out; done
for i in {1..500}; do ./a.out; done | sort | uniq -c
```

If you don't notice something weird, then change 500 to say 1000. Your task is the following.

1. Figure out what is causing this weird behavior.
2. Solve the problem using a mutex.

To check that you fixed the problem by running your code repeatedly as above.

Monitors

Once you have completed this first part, download `cond.c` and read the source code. There is an "assigner thread" and a "reporter thread" in this code, and the goal of the program is for the reporter thread to report the change that the assigner thread has made to a global variable once that change has occurred. If you run this program many times though, you'll see that it doesn't report this change correctly (i.e., "100" will be printed most of the time).

1. Fix this code using mutexes and condition variables.
2. Run your code multiple times to check your solution.

A minor (but sometimes major!) technical note: when we use a condition variable in POSIX, we need to mindful of the so-called *spurious wakeup*. Sometimes multiprocessing systems send hardware signals that cause condition variables to wake up even though they are not supposed to, which gives undefined behavior. If you think this might be happening to you, try using a while loop and an additional boolean variable as a flag to help prevent us from waking up when we are not supposed to. Click here for more details.

[1] For thoroughly testing your solution to the multithreading assignment, you might find nested for loops in bash to be helpful for testing the many command-line arguments that may be passed to your program.

Producer-Consumer

For the last part, download and read `procon.c`. It is a broken toy producer/consumer code, where we aren't actually producing or consuming anything, but simulating this through the use of a counter that counts the number of elements in our hypothetical list. Fix this code using a semaphore, and run your code repeatedly as above to test your solution.