CS333 Spring 2016

Lab 7 (Take Home)

Goal

The goal of this lab is to write simple code to understand race conditions, and the need for locking during concurrent access of shared data.

Part A: A multi-threaded program with race conditions

First, you will write a program `races-partA.c`. The program must initialize a variable `count` to 0. It must then start N concurrent threads. Each thread must run a loop to increment `count` K times, and terminate. The main process must join all N threads, print out the final value of `count` and exit. Ideally, we would expect the final value of `count` to be $N * K$, after $N * K$ increments. But the value printed by your code may not match this ideal value due to race conditions. In addition to the final value of `count`, your program must also print the time elapsed between starting the threads and joining the threads. That is, you must print the time taken by all threads to complete their increment operations. After writing your code, answer the following exercises in the report.

1. Run your program for $N = 10$ and $K = 10,000$. Do this 3 times. Report the average value of `count` and the average time taken for the increments, averaged across all 3 runs.

2. Does your final value of `count` match the ideal value of $N * K$ in any run? If it does not, explain why by pointing out the line(s) in your code that are responsible for the race condition. If it matches, explain why no race conditions occurred.

Part B: A software-based lock

Modify your program, and write `races-partB.c`. You must now try to remove the race condition by using a software-only lock. Replace the code incrementing `count` by the following code. This code uses a variable called `locked`, initialized to zero. All threads now increment count by trying to lock this variable, as follows.

```
while(locked); //do nothing; busy wait
locked = 1; //set lock
count++; //increment
locked = 0; //release lock
```

Repeat the exercises in part A with this modified code. Explain the differences from part A in your answers.

Part C: Using mutexes

Now, modify your program, and write `races-partC.c`. You must now use mutex from the `pthreads` API to ensure correct locking. For example, your threads would increment count as follows.

```
pthread_mutex_lock(&my_mutex);
count++; //increment
pthread_mutex_unlock(&my_mutex);
```

Repeat the exercises in part A with this modified code. Explain the differences from part A in your answers.

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf`, which contains your answers to the exercises above. Be clear and concise in your writing.
- `races-partA.c`, `races-partB.c`, `races-partC.c`.

Evaluation of this lab will be based on reading your code and your report.