CSCI-GA.3033-017 Special Topic: Multicore Programming

Homework 2

Due October 22, 2018

Please solve the following and upload your solutions to your private GitHub repository for the class as homework2.pdf by 11:59pm on the due date above. If for some reason this poses a technical problem, or you wish to include diagrams that you don't wish to spend time drawing in a drawing application, you may hand in a printed copy (*not* hand-written) at the beginning of class (7:10pm) on the day of the deadline. **Unlike labs, late homeworks will be assigned a grade of 0.**

This homework will give you some extra practice thinking about synchronization and thread safety. It is intended to help you hone the skills you need for Lab 1.

---

1. Warm-up: Provide the **minimal** code for implementing the `lock()` and `unlock()` methods of a mutex with a counting semaphore, wherein you assume the program that calls this implementation always calls `unlock()` only when the mutex is locked and only via the thread that locked it.

2. **In your own words**, describe **one** of the possible causes of the Lost Wakeup Problem, including a scenario that triggers this cause, and how to fix it. Use C++ or pseudocode in your explanation iff you find it necessary.

3. If a machine has 4 cores, how many threads should your program spawn? What characteristics of the hardware, your program's threads, or the other software running on this machine would change the answer?

4. Consider the following code:

```
1     static double sum_stat_a = 0;
2     static double sum_stat_b = 0;
3     static double sum_stat_c = 1000;
4     int aggregateStats(double stat_a, double stat_b, double stat_c) {
5           sum_stat_a += stat_a;
6           sum_stat_b -= stat_b;
7           sum_stat_c -= stat_c;
8           return sum_stat_a + sum_stat_b + sum_stat_c;
9     }
10    void init(void) { }
```

Use a single pthread mutex or std::mutex to make this function thread-safe. Add global variables and content to the `init()` function as necessary.

5. Let's make this more parallelizable. We always want to reduce critical sections as much as possible to minimize the time threads need to wait for a resource protected by a lock. Modify the original code

from question 3 to make it thread-safe, but use three mutices this time, one each for `sum_stat_a`, `sum_stat_b`, and `sum_stat_c`.

6. What criterion or criteria must a correct mutex fulfill? Which of these do you think must be true for a general thread synchronization primitive (e.g., barriers, mutices, other types of locks, condition variables, etc.)?

7. What's busy waiting? What's good and bad about it?