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

Homework 2

Due October 30, 2017

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 (6:20pm) 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. Explain (in your own words!) how Peterson's Algorithm works. Why does it need to be re-written for more than 2 threads? For an extra point or two, feel free to explore exactly *what* changes you would need to make.

2. Running on some specific machine *X* with set hardware and software, how many threads should your program spawn, assuming that it has enough parallel tasks to run to occupy many threads? Think about how the hardware (eg, memory and number of cores) and software (ie, programs already occupying some resources) affect the number of threads you should spawn. Don't overthink this one!

3. If multiple on-die caches (eg, L2 and L3 cache) are higher-latency than an L1 cache, why do we have them anyway? (Hint: how big and slow is main memory? How big and slow is the L1 cache?)

4. To the level of detail you feel is necessarily, explain (a) what the lost wakeup condition is in the context of a thread-safe condition variable implementation (or use) is, (b) and how to avoid it. Use C++ or pseudocode in your explanation iff you find it necessary.

5. Consider the following code:

```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 10000;
int aggregateStats(double stat_a, double stat_b, double stat_c) {
      sum_stat_a += stat_a;
      sum_stat_b += stat_b;
      sum_stat_c -= stat_c;
      return sum_stat_a + sum_stat_b + sum_stat_c;
}
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.

6. 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`.