CSCI-UA.0480-001 Special Topic: Multicore Programming

Homework 3
Due July 11, 2019

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 (5:45pm) on the day of the deadline. **Unlike labs, late homeworks will be assigned a grade of 0.**

This homework is intended to assist in your studying for the Midterm Quiz, so I recommend trying to complete it before the Quiz. Remember, you **must** work on homeworks alone: if you confer with any other students on the homework, either to ask questions or to work together, you must cite those students in your solutions.

1. Thread Safety. Consider the following code, a simplified form of the aggregateStats() function from Homework 2.

   ```
   static double sum_stat_a = 0;
   int aggregateStats(double stat_a) {
       sum_stat_a += stat_a;
       return sum_stat_a;
   }
   void init(void) { }
   ```

   a. This code is not thread-safe. Why?
   b. What would you need to do using mutices, semaphores, or condition variables to make it thread-safe?
   c. Based on the techniques you saw in Lecture 11, use CAS to make it thread-safe.

2. Semaphores can be used to make both mutices and condition variables, but (for example) mutices by themselves cannot be used to create true semaphores. What propert(ies) of semaphores make them able to function as mutices? What propert(ies) of semaphores make them able to function as condition variables (with help)? Hint: consider what mutices need for lock() to work properly, and what condition variables need for wait() to work properly.

(continues on next page)
3. Two functions are created for a big project that handles data from a temperature sensor. `addSample()` adds a floating-point temperature sample to a vector called `samples`, and `computeAverage()` returns the average of all temperature samples so far. To speed things up, `addSample()` precomputes a running sum so that `computeAverage()` doesn’t need to sum all samples up to the present.

The following code is intended to be thread-safe if many instances of the `addSample()` and/or `computeAverage()` functions run simultaneously in many threads. Mistakes have been made. Please identify all bugs in this code, multithreading-related, C++, or otherwise. Please be sure to explain (a) why each bug is actually a bug (for example, provide an interleaving between two or more threads that would cause the bug, for a multithreading-related bug) and (b) how it could be fixed.

```cpp
std::mutex sample_mutex; // protect samples vector
std::mutex sum_mutex; // protect sample_sum
std::vector<double> samples;
double sample_sum;

void addSample(const double sample) {
    sample_mutex.lock();
    if (std::isnan(sample)) {  // Don’t try to keep a NaN sample – sensor not working?
        return;
    }
    samples.push_back(sample);
    sample_mutex.unlock();
    sample_sum += sample;
    return;
}

double computeAverage() {
    sum_mutex.lock();
    return sample_sum / samples.size();
    sum_mutex.unlock();
}
```

4. Consider the Filter Algorithm for exactly two threads. Explain how it would work for these two threads simultaneously trying to get a lock. Compare this result to how Peterson’s Algorithm works, and finally use this to explain how the Filter Algorithm works for 3, 4, and N threads.