Alexander Nguyen

Christopher Mitchell

July 6th, 2019

Homework 3

1.

a. This code is not thread safe because sum_stat_a is a shared variable that is susceptible to a data race with other threads.

b. It is possible to be thread safe by putting mutexes around the and having a local variable to keep track of the sum as such code as such:

```
static double sum_stat_a = 0;
mutex mu;
int aggregateStats(double stat_a) {
        lock(mu);
        sum_stat_a += stat_a;
        double sum = sum_stat_a;
        unlock(mu)
        return sum;
}
void init(void) { }
```

c. The CAS thread safe code is below:

```
static double sum_stat_a = 0;

int aggregateStats(double stat_a) {
        double temp;

        while(true){
                temp = sum_stat_a;
                if(CAS(&sum_stat_a, temp, stat_a + temp)) break;
        }

        return stat_a + temp;
}

void init(void) { }
```

2.
Properties of semaphores to function as mutexes:

- Semaphores are able to function as mutexes by protecting a critical section through means such as a binary semaphore. Semaphores also have the property of being starvation and deadlock free similar to a “blocking-mutex”.

Properties of semaphores to function as condition variables:

- By using a queue, threads can hold a private semaphore that decrements to zero and be enqueued if they are waiting for a condition predicate. Once the condition predicate is met, it is possible to simulate both signal or broadcast to all queued threads by incrementing one or all the thread semaphores in the queue and dequeuing the respective thread.

3.
Edits commented in the code are as follows:

```
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?
//The lock is not released when returned early. Simply release the lock before returning.
 return;
 }
 samples.push_back(sample);
 sample_mutex.unlock();
 sample_sum += sample;
// sample_sum should be added within the lock, simply move this code to before the
lockwas released.
 return;
}
double computeAverage() {
 sum_mutex.lock();
 return sample_sum / samples.size();
//This division can lead to issues if another sample is added, can be fixed using a local
variable and additional sample_mutex_lock to ensure no additional samples are added
and the values returned are from an unchanged local variable.
 sum_mutex.unlock();
}
}
```

A fixed copy of the code is represented below:

```
        sample_mutex.lock();
        if (std::isnan(sample)) { // Don't try to keep a NaN sample –
                sensor not working?
                sample_mutex.unlock(); //Release the lock before returning early
                return;
        }
        samples.push_back(sample);
        sample_sum += sample; //sample_sum should not be incremented outside the lock
        sample_mutex.unlock();

        return;
}
double computeAverage() {
        sum_mutex.lock();
        sample_mutex.lock();
        //A lock to prevent adding samples is required to ensure size and sum are not
        changing mid-way through the function.
        double average = sample_sum/samples.size();
        //Store the average in a local variable since directly returning the division between
        sum and size may  change outside the lock
        sample_mutex.unlock();
        sum_mutex.unlock();
        return average;

}
```

4.
Peterson's Algorithm:
For two threads, this algorithm ensures mutual exclusion by having flags to signal who is trying to access the lock and a victim variable to signal who is next to get the lock. If a thread was once a victim but no longer is, it will then be able to retrieve the lock. If the thread is the victim but no other thread is currently seeking to retrieve the lock, then that thread will be granted the lock regardless of being the victim.

For 3 threads:
There will be n-1 = 2 levels in the filter algorithm. The filter algorithm works as such: while the thread is currently the victim and there are threads at higher levels, then the thread must wait until either a new thread becomes a victim at that respective level or there are no more threads in further levels. Once a thread retrieves a lock, it must return to the top level if it intends to retrieve the lock again. For this specific case, Peterson's algorithm occurs at the highest level (2) while level 1 serves as a waiting room. At level 1 there is a max of 2 threads, level 2 has 1 thread at max respectively,

For 4 threads:

There will be n-1 = 3 levels. The first two serve as waiting rooms while level three will act like Peterson's algorithm. The mechanism will be similar to the one above. At level 1 there is a max of 3 threads, level 2 has 2 threads, and level 1 has 1 thread at max respectively.

For n threads:
There will be n-1 threads. At each level there is at most n-(1*level) threads where the highest-level acts like Peterson's algorithm. The mechanism is similar to the algorithm described above.