Will Suitor
CSCI Multicore - HW 2

1. A big difference between semaphores and condition variables is that semaphores rely on memory. This means that when someone posts something to the semaphore, there does not need to be someone actively waiting to receive it because the semaphore is simply incremented in shared memory. On the other hand, condition variables do not rely on memory. With a condition variable, a signal can be lost if there is no one waiting for it. What this means is that the consumer can be waiting when there is actually something to consume because it started waiting after the signal was sent.

Another difference is how semaphores and condition variables wait. Semaphores are busy-waiting while condition variables are not. Semaphores keep a thread active while checking to see if some other thread has called up() while condition variables put a thread to sleep and then are only woken up when they receive a signal. So, semaphores continually check to see if there is something to process while condition variables only wake up a thread when there is definitely something to process.

2. Mutices must meet three criteria. They must guarantee mutual exclusion, be deadlock-free and be starvation-free.

Semaphores can be used to guarantee mutual exclusion. For example, a binary semaphore could be used as a valid mutex. When it was 1, the lock would be available and when it was at 0, the lock would be unavailable. However, they are more used to order the processes in threads (producer-consumer). Semaphores are not necessarily deadlock-free. If all the threads operating call down() when it is at 0, then all the threads will be suspended. However, this model does not make practical sense because it’s a model where there are just consumers without producers. So, in practical cases, I would say semaphores are deadlock-free. Semaphores are starvation-free, assuming data is constantly being provided. The only caveat being if there is only a small number of producers and a ton of consumers. If there are a lot of consumers waiting in the down() state, then when up() is called, there is a race for the data. This means that one thread could end up waiting for a long time if it keeps getting beaten to the data.

Again, condition variables can be used to guarantee mutual exclusion but are more meant for synchronization between threads. Condition variables are more used for threads to say that a certain thing has to happen before another thread can proceed, rather than just guaranteeing only one thread at a time can write to a certain address/addresses in memory. Condition variables are not necessarily deadlock-free. If thread A is waiting for a signal from thread B, there is nothing stopping thread B to then wait itself for a signal from thread A. In this case, both threads will then be suspended waiting for a signal from the other one. In the same way as semaphores, I would say condition variables are starvation-free, bar the case where there are too many consumers and too few producers.

3.

```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std::mutex mtx;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
        double temp = 0;
        mtx.lock();
        sum_stat_a += stat_a;
        sum_stat_b -= stat_b;
        sum_stat_c -= stat_c;
        temp = sum_stat_a + sum_stat_b + sum_stat_c;
        mtx.unlock();
        return temp;
}
void init(void) { }
```

I’m using the temp variable here because I’m assuming you want the aggregate of the stats at the moment the function is called, not when the function is returning. For example, if I just used the lock around the operations that alter the variable, it’s possible another function would then alter them before the function returned the sum of the three globals.

4.
```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std::mutex mtxA;
std::mutex mtxB;
std::mutex mtxC;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
        double temp = 0;

        mtxA.lock();
        sum_stat_a += stat_a;
        temp += sum_stat_a;
        mtxA.unlock();

        mtxB.lock();
        sum_stat_b -= stat_b;
        temp+= sum_stat_b;
        mtxB.unlock();

        mtxC.lock();
        sum_stat_c -= stat_c;
        temp += sum_stat_c;
        mtxC.unlock();

        return temp;
}
void init(void) { }
```

This reduces the amount of time other threads would spend waiting if they just needed one of the globals. However, unlike the last one, I cannot guarantee that it will return the value of the aggregate at the moment of call. For example, B and C can be altered while A is locked. So, using one or three locks really comes down to preference or the specific context that the function is being used in.

5. The root of the lost wakeup problem is that condition variables do not rely on memory. So, if a function sends the signal while no other threads are waiting, that signal is just lost because it will not be recorded in memory anywhere. Then, when another thread begins to wait, it will wait while there is actually something to process because the signal was sent before it started waiting. The way to fix this would probably be to include a semaphore, which would actually be

incremented in memory every time a signal is sent out. Therefore, the consumer thread could first check the semaphore before beginning to wait for the signal.
6.
A.
```
class Queue{
std::mutex mtx;
std::condition_variable cv;
item head;
item tail;

void push(item){
        std::unique_lock lock(mtx);
        if(this.isEmpty()){
                lock.lock();
                this.head = item;
                this.tail = item;
                lock.unlock();
                cv.push_one();
                return;
        }
        lock.lock();
        this.tail.next() = item;
        this.tail = item;
        lock.unlock();
        cv.push_one();
        return;
}
item pop(){
        std::unique_lock lock(mtx);
        lock.lock();
        if (this.isEmpty()){
                lock.unlock();
                return NULL;
        }
        item temp = this.head;
        this.head= this.head.next();
        lock.unlock();
        return temp;
}
item listen(){
        std::unique_lock lock(mtx);
        while(this.isEmpty()){
                cv.wait(lock);
        }
        lock.lock();
        item temp = this.head;
        this.head = head.next();
        lock.unlock();
        return temp;
}

};
```

B. Only Semaphores:

```
class Queue{
item head;
item tail;
Semaphore queueCount(0); //initial count of 0
Semaphore lock(1); //initial count of 1

void push(item){
        if(this.isEmpty()){
                lock.down();
                this.head = item;
                this.tail = item;
                lock.up();
                queueCount.up();
                return;
        }
        lock.down();
        this.tail.next() = item;
        this.tail = item;
        lock.up();
        queueCount.up();
        return;
}
item pop(){
        lock.down();
        if(this.isEmpty()){
                lock.up();
                return NULL;
        }
        item temp = this.head;
        this.head = this.head.next();
        queueCount.down();
        lock.up()
        return temp;
}

item listen(){
        queueCount.down();
        lock.down();
        item temp = this.head;
        this.head = this.head.next();
        lock.up();
        return temp;
}
};
```