1.a) This code is not thread-safe because we don’t have a lock around the critical section, which means that sum_stat_a (a global variable that can be modified by another thread also running aggregateStats) could’ve been changed by another thread before returning.

b) You can have a mutex lock around line 3, where we update sum_stat_a and also create a local variable within the function and return that instead of sum_stat_a because this local variable won’t be changed by another thread. On the other hand, when we unlock to return sum_stat_a another thread may have changed sum_stat_a and we don’t really know what we return.

```
static double sum_stat_a = 0;
std::mutex mut;
int aggregateStats(double stat_a) {
    std::unique_lock<std::mutex> lock(mut);
    sum_stat_a += stat_a;
    double sum = sum_stat_a;
    lock.unlock();
    return sum;
}
```

c)

```
static double sum_stat_a = 0;
int aggregateStats(double stat_a) {
        while (true) {
                double sum = sum_stat_a;
                double new_sum = sum + stat_a;
                if (CAS( &sum_stat_a, sum, new_sum)) {
                        break;
                }
        }
        return new_sum;
```

2. For a mutex lock to work properly, we need to make sure that it’s mutually exclusive, starvation free and deadlock free. Because semaphore is a non-negative integer value, we can use it to solve the mutual exclusion problem that the mutex lock is required to solve. We can have a semaphore that is initialized to 1 and will alternate between the value of 1 (unlocked) and 0 (locked). When trying to lock, we call sem_down() before the critical section and sem_up() corresponds to unlocking. It is starvation free because as long as the right thread calls sem_up() and increments the mutex semaphore value to 1, nothing can block this anymore and the lock will be unlocked and other threads will be able to execute sem_down. In the case of the implementation in the slides where we have two semaphores to implement a mutex, we can guarantee that it’s deadlock free because of the second semaphores that protects the critical section, whereas we always call sem_up() before leaving the critical section. As long as our semaphore is initialized to a value above 0, we can guarantee it’s deadlock free.

An important property of condition variable for wait() to work properly is that the threads are able to wait without locking the related function. To implement wait() with semaphores and a mutex, we can have a condition semaphore to protect the queue of waiting threads whenever we enqueue a waiter and a mutex to lock the general state, as well as a semaphore initialized to 0 within the thread to wake and put the thread to sleep. After enqueuing the new waiting thread into the queue of waiters, we would release the general mutex and put the thread to sleep by calling sem_down() on the private semaphore within the thread, satisfying the property of wait. Once the thread is woken up by signal(), then we can reacquire the thread. With help of a queue of waiter, we can also guarantee linearizability for wait() and starvation-free. Since sem_down() can't block sem_up(), we should also have proper wait() whenever we sem_up().

Criteria for wait():

Linearizability – no lost wait()

Proper wait(), no spurious wakeups – i.e a thread waking up even though the condition Not true and supposed to be waiting

Starvation-free : FIFO wait()ing

Deadlock-free: wait() must not prevent signal()

3.

addSample bug(s) :

1. When we return from the if(isnan(sample)) statement, the code doesn't unlock the mutex first before returning and we'll be stuck in a deadlock. For example, if thread1 tries to addSample with a sample that is NAN, and then thread2 also tries to add a sample, since thread1 now has the mutex locked and returned without unlocking it, thread2 (and future threads) will always be stuck and can't enter critical section of addSample. We can fix by adding sample_mutex.unlock() before returning.

2. There's no lock around the sample_sum and so when we try to update sample_sum we can't update because sum_mutex might be locked by computeAverage. Also another thread may try to change sample_sum.

computeAverage bugs:

1. Didn't lock the sample_mutex which can affect the average because another thread might have added a sample into the samples vector but haven't updated the sum yet and technically can't update the sum(because computeAverage has locked the sum_mutex) . For example, thread1 is in computeAverage and meanwhile thread2 is in addSample. Thread2 successfully pushed back another sample into the samples vector, increasing the size by 1. Because thread1 has the sum_mutex locked thread2 can't update the sample_sum even if it happens before thread1 tries to return sample_sum/samples.size(). So thread1 will return an average that is incorrect because the sample_sum doesn't correspond to the number of elements.
2. Having the return sample_sum/ samples.size() inside the lock and unlock will cause a deadlock because we return with the lock still being locked. This way no other thread can change the sample_sum.

3. Both sample_sum and the samples vector are global variables so if you return them from a function you can't guarantee what is being returned as other threads may have changed their values.

Fix:

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

void addSample(const double sample) {
        sample_mutex.lock();
        sum_mutex.lock();
        if (std::isnan(sample)) {
                sample_mutex.unlock();
                sum_mutex.unlock();
                return;
        }
        samples.push_back(sample);
        sample_sum += sample;
        sample_mutex.unlock();
        sum_mutex.unlock();
        return;
}

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

4. 2 threads want to lock at the same time.

Filter Algorithm :
We have initial state: int victim[1] = { -1 } // we have 1 level since n = 2
                       Int level[2] = { 0 , 0}

Both threads will try to enter and get the lock at the same time so they will both try to enter level 1 (the for loop in lock). At least one will be able to enter (either A or B) and the other one will be blocked from entering level 1. Because of cache coherence only one thread can access the memory address of the arrays, one of the thread will be able to set level[i] = L and victim[L] = I first. The second thread will try to set level[i] = L to indicate that it wants the lock and set itself as the victim, but will be stuck in the while loop. At this point the first thread is able to skip the while loop and get the lock because there's another thread trying to be in the same level. Once the first thread finishes the second thread can enter.

Peterson's Algorithm:
Because of cache coherence, both threads can't write to the same address in memory at once. So the victim variable act as a tie breaker and will dictate who wins. Both threads will set their flags to be true to indicate that they want the lock. The thread that sets victim = me later will be invalidated because it will be stuck in the while loop, and will have to wait until the winning thread calls unlock.

Filter Algorithm on 3+ threads:
One thread will be able to enter the first level, and this thread will wait until there is another thread trying to enter in order to proceed into the next level. The second thread that enters will be in the first level, and will only proceed until another thread tries to enter the level that the second thread is in and the thread before it moves forward. This just keeps happening until one lock reaches the the last level and is able to grab the lock. Once it's finished with the lock, the next thread waiting in line will be able to proceed.