Will Suitor
HW #3

1. A. This code isn't thread-safe because any thread could read/write sum_stat_a without a lock around out. Essentially, there is an unprotected variable that is shared between multiple threads.

B. If you wanted to make it thread-safe, you could simply put a lock guard as the first line of the function.

C. If you wanted to use a Compare-And-Swap to make this thread-safe, you could do:

```
static double sum_stat_a = 0;
int aggregateStats(double stat_a) {
        while(true){
                tempSumA = sum_stat_a;
                if(CAS(*sum_stat_a, tempSumA, tempSumA + stat_a)){
                        return tempSumA + stat_a;
                }
        }
}
```

2. Semaphores can be used as mutices pretty easily. Mutices need to provide lock() and unlock() methods that are mutually exclusive, deadlock-free, and starvation-free. To create mutual exclusion, you just need a semaphore initialized to 1 (binary semaphore). When a thread calls down on that semaphore, that is the equivalent of a thread calling lock() in a mutex. The semaphore will be set to 0, so if another thread calls down, it will be stuck waiting until the other thread calls up() or the equivalent of unlock(). They are deadlock-free, assuming they are used correctly. As long as a thread using a binary semaphore doesn't call down() twice in a row, there should not be a deadlock. Finally, it is starvation-free. When the holding thread calls up(), any thread/s that are waiting can see that the lock is available and race to get it.

Similarly, semaphores can be used as condition variables with some minor changes. Semaphores are FIFO, so up() is more or less equivalent to signal(). wait() would be similar to down(). When the condition variable is not met, the semaphore has to wait in the down state and is awoken when another thread calls up().

In terms of properties, semaphores are starvation-free, deadlock-free, and linearizable. All those qualities allow for them to be used as condition variables and mutices.

3. In addSample(), the sample_sum could be incorrectly altered. Sample_sum is a global variable and the '+=' operator is actually two operations. This means that if two threads run through addSample() at the same time, they could both read the same initial sample_sum.

Then, one writes that the sum of their sample and that initial value to sample_sum. Then, the second thread who calculated their sum with the old sample_sum will overwrite it. The new sum will no longer include the first sample. To solve this, just surround this with the sum_mutex lock() and unlock().

In computeAverage(), you're going to create a deadlock. The function returns before it releases the lock, meaning it is never released. To fix this, create a temporary variable and set that equal to 'sample_sum / samples.size()' within lock() unlock(). Then, return the temporary variable after unlocking.

A small thing (not a bug necessarily) is that the NaN test in the addSample() should happen outside the mutex. There is no need to lock during that test.

4. Filter Algorithm for two threads is relatively simple. There will only be one level (L1). When both threads try to enter at the same time, one tries to enter L1 first and sets the victim of L1 to itself. Then, the other thread sets the victim to itself and attempts to enter L1. This allows the first thread to proceed into L1 while the other is stuck waiting. When the first thread leaves L1 (unlocks), the other thread is free to proceed into L1 (get the lock).

Peterson's Algorithm would handle two threads trying to get the lock at the same time differently. Peterson's Algorithm utilizes a global array of boolean 'flag' variables and a global int victim. First, both threads would figure out whether they correspond to 0 or 1. Then, both threads would attempt to set the victim to themselves. Whoever sets it first is then overwritten by the second thread. That thread can then proceed while the other thread is stuck checking if the other's flag is still true. When the first thread unlocks, it sets its flag to false and the second thread can proceed.

Filter Algorithm for three threads works similarly. If threads A, B, and C try to get the lock at the same time, then first thread A tries to enter L1. Since there is no thread at a higher level, it succeeds in entering L1. Then, thread B tries to enter L1. When the victim is updated, thread A proceeds to L2 (and gets the lock). When A moves, B enters L1. When C tries to enter L1, it gets stuck outside waiting outside. This prompts B to attempt to enter L2 and it will enter when A leaves (unlocks). When that happens, C enters L1 and waits until B leaves to enter L2.

It's very similar for four threads. There will be three levels (L1, L2, L3) and four threads (A, B, C, D). If they all try to enter the lock at the same time, A will enter L1 first since there are no threads below it). Then, thread B tries to enter, sets the victim of L1 to itself, causing A to enter L2. B is stuck outside until C tries to enter L1. When that happens, B enters L1 and tries to enter L2, forcing A into L3 (gets the lock). This pattern repeats as A unlocks, B gets it, then C, then D.

Then, for n threads, it will follow that pattern. Threads will continually try to enter the level ahead of it and only be able to proceed into that level if they are no longer the victim of their level or there are no threads in the levels ahead of them.