1. A) The part of code that modifies the shared variable is not wrapped in a critical section. There is a possibility that another thread modifies aggregateStats between line 3 and 4.

   B) We can use mutices to create a critical section around line 3, capture the state of sum_stat_a in another temporary variable and outside the critical section return the variable.

   C) int aggregateStats(double stat a) {

   ```
   While ( true ) {
       Int temp = sum_stat_a;
       If ( CAS(temp , sum_stat_a ,   sum_stat_a + stat_a ){
           Return temp;
       }
   }
   }
   ```

2. By initializing a semaphore to 1 it guarantees exclusion of all other threads except one allowing it to work as a mutex. When a thread tries to down ( ) a semaphore that is less or equal to zero, the thread is suspended similar to a wait( ) call on a condition variable. This property of semaphores allow it to function as a condition variable.

3. 1) If the parameter sample actually turns out to be NaN in on thread it doesn't unlock sample_mutex before it returns causing a deadlock. The fix is simple as unlocking sample_mutex before the return statement inside if (std:: isnan(sample))

   2) Similar to bug 1, calling computeAverage( ) unlocks after the return statement The fix is having a temporary variable to hold the result of sample_sum / samples.size( ) and return that variable after unlocking.

   3) A very simple non-realistic example of a bug would be adding one sample and trying to compute the average. If sample_sum += sample is not inside the critical section there is a possibility that the computed average will still remain zero. Thus we want sample_sum += sample to also go inside the critical section.

   4) There is a possibility that computeAverage will throw a divide by zero exception. We want to take care of the case where there are no elements in the samples vector with a separate if case.

4. Peterson's algorithm has a shared variable victim which indicates the thread that is trying to acquire the lock. If two threads attempt to get the lock simultaneously victim will be set to either one of the threads. Array flag[2] will contain two trues and victim will indicate one specific thread. In one thread, once it reaches the while loop it will wait until the victim is no longer itself.

   In the filter algorithm a thread can advance to the next level only if a new thread attempts to join the level and updates victim[L] OR there are no threads in a level higher or equal to the level (from level 1 not 0) that the current thread is at. In this algorithm the lock is given to the thread that reaches the top level first. If two threads attempt to acquire the lock the lock that first succeeds to advance to level 1 will eventually acquire the lock first.

   The filter algorithm is also applicable to more than two threads. The first thread that tries to advance to level 1 will easily advance to the next level. Even if another thread joins the same level the victim of that level will have been updated to the new thread allowing the current thread to advance to the next level. So again the first thread to advance to level 1 will eventually acquire the lock first. For the rest of the threads the condition that "there are no threads in a level higher or equal to the level" cannot be met until one thread acquires the lock and leaves. Thus the only way other threads can advance to the next level is if another thread attempts to join the current level they are at and updates victim[L]. So the second thread that attempts to join level 1 will only join once the third thread attempts to join level 1. So on and so forth.