1. What accounts for the inconsistency of the final value of the count variable compared to the sum of the local counts for each thread in the version of your program that has no lock/unlock calls?

   To prevent other threads that created increment the count accidentally during the runtime.

2. If you test the version of your program that has no lock/unlock operations with a smaller loop bound, there is often no inconsistency in the final value of count compared to when you use a larger loop bound. Why?

   A smaller loop can cause less race conditions to happen.

3. Why are the local variables that are printed out always consistent?

   Threads will have their own memory space in the runtime, so this will keep local variables from being accessed by other threads.

4. How does your solution ensure the final value of count will always be consistent (with any loop bound and increment values)?

   We only allow one thread to modify the count at a time by using locks.

5. Consider the two versions of your ptcount.c code. One with the lock and unlock operations, and one without. Run both with a loop count of 1 million, using the *time* command: "bash> time ./ptcount 1000000 1". Real-time is total time, User time is time spent in User Mode. SYS time is time spent in OS mode. User and SYS time will not add up to Real for various reasons that need not concern you at this time. Why do you think the times for the two versions of the program are so different?

   The one using lock/unlock is significantly slower because it only allows one thread run the count function at a time. However, all thread are running concurrently on the one without lock/unlcok.