Lab 5 WriteUp

Nicholas Jurden 2415098

1. Because threads exist within the same process and share state, memory, address spaces, among other things, variables modified by threads may experience concurrency issues that produce values that arent expected.

2. As the slides state,

   > *Probability of inconsistencies in shared data, such as count, depends on…the number of times threads execute the code operating on shared data.*

   In this case, because the `count` variable is global, the more that `inc_count` is called increases the probability of threads interleaving in ways that “loses” the result of one of the threads incrementing `count`. Threads individual counts return correctly in part because they are locally declared variables.

3. In order to avoid “using” parts of a thread that increment the global `count`, we use `pthread_mutex_lock` before the section of code that updates the global variable and `pthread_mutex_unlock` after the global variable is updated. This ensures that within the “critical section” as indicated by the lock, other threads cannot enter while one thread is executing. This preserves the address space for one thread at a time and ensures no increments are lost.

4. My intuition would say that, without locks, the system must take a fraction of time to manage threads entering and exiting the address space and other shared resources of the process when incrementing the global `count`. It would make sense that context switching required by interleaving threads would result in additional time required to execute.