Jay Offerdahl
EECS 678
Lab 5 Report

Lab 5: Pthreads

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?

- Without any lock/unlock calls, each thread may try to modify the global count variable at the same time, meaning one, two, or even three increments may be attempted, but only one can get through at a time because the threads were interleaved. This results in a huge loss of intended increments to the count.

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?

- This happens because there are simply less race conditions than that of a larger loop bound. Each time a race condition happens an increment or two is lost. With a smaller loop bound, some threads may finish before others are even created simply because it takes time to create the thread. For example, we could enter the thread creation for loop in main, create a thread, and in the time that it takes to increment the for loop, check the condition, and create another thread, the first thread might have already made significant uninterrupted progress in its process. It's for this reason that smaller loop bounds are less inconsistent in the final value.

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

- The local variables printed out in each thread are always consistent because threads only share global variables. Since the variable tracking the local count is stored in "loc" which is locally declared inside of *inc_count, only the current thread has access to it. This is an especially important consequence of threading as it allows us to distinguish threads more than we already do.
    - Concerning the consistency between the two version of the program, the version that doesn't lock and unlock the mutex still prints out a "full" count for each thread because even though the race condition exists for the global variable, there's no race condition for locally declared variables since they're all separate.

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

- My solution ensure that the final value of count is always consistent no matter the loop bound/increment values by implementing a mutex. This means that each thread must lock the mutex before proceeding and modifying the count variable in the critical section. Once the operation is complete, the thread unlocks the mutex, and the next ready thread is able to then lock the mutex and complete its operation. This happens until all threads exit.

Jay Offerdahl
EECS 678
Lab 5 Report

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* 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?

- Without lock/unlock operations:
  - Real: 0.04s
  - User: 0.02s
  - Sys: 0.003s
- With lock/unlock operations:
  - Real: 0.247s
  - User: 0.299s
  - Sys: 0.419s
- The times for these two versions are so different because in the first version, with no locking mechanism, the loops in each thread run without delay, going as fast as they can to finish the process. In the version of the program that locks and unlocks the resource, time is spent waiting for the other threads to wait. In short, the second version waits when the first version wouldn't because the threads in the second version respect the critical region.