EECS 678 Lab 07 Report

Author: Jace Kline 2881618

Questions

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?

In the version of the program with no lock/unlock calls, the C statement `count = count + my_args->inc` is not an atomic operation. In fact, this statement will result in a sequence of 3 instructions {MOV, ADD, MOV} on x86 hardware. The reason that this will cause inconsistency in the final count is as follows: The count is a global variable to the program, and hence is shared by all threads (in the same process address space). With multiple threads accessing and modifying the data stored in count, what will happen on a system with preemptive scheduling is that the scheduler could interrupt the process running this multi-threaded program. When this process starts running again, a different thread could be scheduled from the previous one running, regardless of which atomic instruction the previous thread was running in this process. Now, suppose that thread 1 was interrupted after the initial MOV assembly operation when trying to update the count variable. In other words, it has read the initial value of count into a register, then gets preempted by the scheduler, but it did not have time to increment count. Now, when this process starts running again on the CPU, suppose another thread (thread 2) starts running and enters its critical section to update the count variable. Thread 2 will read in the same value that thread 1 read in and then increment it. This results in count = increment(count) for some increment. This is where the problem lies. When thread 1 runs again, it will resume its atomic operation sequence where it has already stored the previously read value of count into a register. Then it will increment count from the old value of count (not count that was updated by thread 2). Hence, after 2 increment operations coming from different threads, the actual value of count will be count = increment(count) where the expected output should be count = increment(increment(count)). If this happens multiple times over a large iteration range, then the actual final value of count could be much lower than the expected final value of count due to the propogation of the low value.

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?

With a low number of iterations, the probability of interleaving of critical sections between the multiple threads is much lower because there are simply fewer opportunities for the process to get interrupted at the exact spot where this problem would occur (between the MOV and ADD instructions). However, it is still possible for inconsistency to occur.

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

The local variables in each thread are consistent because each thread gets its own stack memory region within the process and all local variables to each thread are stored in their own respective stack. In other words, each thread has its own version of the local variable to mutate (despite having the same name), and no thread ever accesses/mutates another thread's local variable.

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

Both solutions that I implemented ensure that no two threads can ever enter their critical sections at the same time. This is important because the critical sections of the threads are when they are accessing + updating shared data and ultimately when the possibility for arbitrary interleavings of atomic instructions could occur that result in inconsistent behavior. In the first solution (ptcount.c), I used the functionality of a pthread's mutex to lock and unlock the shared/global mutex around the critical section of each thread. In the second solution (ptcount_atomic.c), I used the __atomic_add_fetch(...) GCC built in function that performs an increment operation in one atomic step, and therefore prevents the possibility of interleavings.

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?

In version 1 (ptcount.c - using pthread's mutex solution), the user time was 33.71 and the system time was 19.44. In version 2 (ptcount_atomic.c - using GCC builtin atomic add), the user time was 16.23 and the system time was 0. Version 1's time taken was much larger than version 2's overall. This is mainly because in version 1, the pthread_mutex_lock(...) requires the operating system to block the calling thread until the lock is received. This means that not only the calling thread has to be blocked, which spends time in user mode, but also that there is an overhead of using the operating system (system-level control) to handle the mutex. In the second solution, however, the use of the GCC atomic add instuction does not require any system-level control and additionally there is no blocking that occurs. Hence, the version 2 solution was much more time efficient.