CMSC 23010: Homework 3a Design Documentation

Will Thomas

November 3, 2022

Introduction

The goal of the following design documentation is to outline a cohesive plan for the design, correctness testing, and performance analysis of a program that will show the performance differences of the following locks: TASLock, TTASLock, PTHREAD_MUTEX_t, and ALock.

Design

The program will implement functions to run two tests regarding concurrent performance.

The first test is the work-based counter test. Essentially, we will time how long it takes to count to a certain large number. For the purposes of this assignment, that large number will be 750,000. I believe this to be a sufficiently large number to limit case to case variance while also not being so large as to lock other students out of SLURM, although I will change the number as I see fit if this assumption proves to not be true.

The second test tests lock fairness, especially for TASLock, TTASLock, and PTHREAD_MUTEX_t, as ALock assumes a high degree of fairness as it is FCFS. We will test fairness by having an outfile for each thread being tested (or maybe a thread-specific counter). We will then plot frequency in the critical section in a histogram, as well as other techniques of analysis to better understand fairness.

Implementation-wise, this program will have the concurrent and serial counter functions as defined in the assignment handout as well as the locks as defined in the textbook. Seeing as the fairness tests will be utilizing the same code, I will pass an extra variable to the concurrent function that acts as a flag on whether or not we should record number of times in critical section per thread. Obviously this treatment is not needed for the serial version of the counter seeing as one thread is doing all of the work.

Correctness Testing

Correctness testing is a little tricky for this assignment, especially with ALock. For TASLock, TTASLock, and PTHREAD_MUTEX_t, we need to ensure that the big number we are counting to is what is held in the counter after the program is run. We can test this by running each version a large amount of times until we can be sure beyond any reasonable doubt that there is never a case in which more than one thread is in the critical section at the same time, as that is the main invariant for these tests.

ALock will prove to be more complicated, especially because it must uphold a FCFS condition. Therefore, beyond just counting to the correct number (the mutual exclusion condition), we must show that the queue works correctly. We can do this by, in another program, implementing the same ALock that we intend to implement in the main function. From there, in a for loop, we have the main thread at each iteration:

1. Create a new Thread

2. Have that thread enter in a function where it immediately calls for the lock. Once it enters the CS, sleep for an arbitrary but short amount of time to build up contention and then print its tid in an outfile before releasing the lock.

If the thread ids in the outfile are in increasing order, we can reason that the queue upholds the FCFS condition. This is because we can control the order in which the threads call the lock, and we can manufacture more contention through random sleep periods. With a large enough amount of threads and trials, we can show that the invariant is upheld.

Proving the correctness of the counter is trivial; the only invariant to be upheld is that the counter does not skip any values and that it ends when it should. We can show the former condition by having the thread in CS print out the value of counter and then ensuring that everything goes in the correct order. For sufficiently large numbers to count to, this can be verified via a python script. The latter condition is verifiable via the same method; must make sure the last number printed out is the number we intend to count to.

Performance Hypothesis and Testing Plan

1. **Idle Lock Overhead**

   It is important to note that we will run each experiment 6 times and take the median as the final value.

   (a) TASLock

   I predict that TASLock will have the smallest overhead, leaving it only marginally longer in total computation time than the serial version of the code. I think that the reason for this is that it takes advantage of hardware. Also, because there is only one thread, it will avoid all of the performance drawbacks inherent in using the TASLock regarding busing issues and cache misses.

   (b) Mutex

   Performance here depends on exactly how the pthread_mutex_t is implemented on the machine in which the code is run. We can assume that it is correct and portable and probably quite fast, so I would assume the overhead difference would be quite small, much of which would come from creating the single pthread.

   (c) ALock

   I hypothesize that ALock will have the most substantial overhead. This is due to all of the infrastructure that it needs to work correctly, most notably the array.

   (d) TTASLock

   I believe that in the case of only one thread being used, TTASLock will perform almost identically to the TASLock.

2. **Lock Scaling**

   (a) TASLock

   I predict that TASLock will have the worst performance out of any of the locks. This is simply something that we have already seen, and we already know that the reason is because of issues with cache misses and busing updates.

   (b) Mutex

   I predict the mutex lock to have the best performance of any of the locks tested, simply because it is probably very fast to be implemented as the base lock.

   (c) ALock

   I suspect that ALock will have the second best performance behind the mutex lock, simply because of the resolved cache-coherence traffic (especially if we implement padding).

(d) TTASLock

As we know and have seen in the lectures and textbook, the TTASLock can be expected to perform better than the TASLock but not as good as the ALock. This is because although a lot of the cache update issues have been obviated, there is still the collective miss that occurs after a thread leaves the critical section (releases the lock).

3. **Fairness**

The test for fairness will be as follows:

We will test the exact same parameters as the Lock Scaling test; this is because I reason that it hits a very sufficient range of inputs.

As mentioned earlier in the document, we will create a histogram for each lock type. For each histogram, the buckets will be one per thread, and the y-axis will be 'number of times in critical section.' My predictions on performance for each lock are as such:

(a) TASLock

There is no defined ordering to the TASLock; it guarantees no conditions like FIFO, but it is deadlock free and obviously upholds the mutual exclusion condition. As such, I predict that it will be semi-random on which thread obtains the lock at each ownership switch, meaning they should all be roughly the same, especially after being run a good many (15) times.

(b) TTASLock

Just like TASLock, it really cannot be known which thread will enter the CS next. As such, I predict that the fairness histogram will be roughly even after sufficient runs.

(c) ALock

Although there are always degenerate effects to break the FIFO assumption, I predict that the Anderson lock will have non-equal counts per thread (assuming a number evenly divisible by n) a statistically insignificant amount of times.

(d) Mutex

Although it is hard to say for certain about how Mutex will perform, I suspect that it will uphold similarly to the ALock, simply because a general lock interface should definitely avoid starvation for overall program performance. As such, I suspect a similar statistically insignificant amount of times that the number of times in the CS differs from thread to thread.