Stefan Knott

27 April 2015

CSCI 4830: Semester Project

Analyzing and Comparing Locking Mechanisms in Varying Systems

INTRO:

For my project, I hope to learn about the drawbacks and benefits of different locking mechanisms. This is of interest to me because today's systems can be very different from one another and it is important to know when it is best to use what. Some systems have a couple threads and high contention for shared resources, some have dozens of threads and no contention. With this much variation of system infrastructure, it is easy to believe that there cannot be just one maximally efficient mutual exclusion implementation for all. I seek to find out what is the most efficient mutual exclusion implementation for various applications. This projects main source code is written in C, and all test scripts written in BASH.

SYNOPSIS:

To learn more about multi-threading locking mechanisms I will test multiple locking implementations. The ones I am going to test are: mutex, spin lock, binary semaphore, and a test-test-and-set lock with exponential back off. I will use the Pthread library for the mutex, spin lock, and binary semaphore, and witll have a TTASB lock implemented in code within the program

To test these mechanisms I will vary the thread pool size CPU set size, and the contention level of shared resources in the program. I test with thread pools of sizes 8, 15, 30, CPU set sizes of 2, 4, 8, and low/high contention levels . I plan to test these varying levels of contention by accessing a global shared variable a small amount of times by each thread (low contention) and by accessing the variable much more times (high contention). The source code in threading_suite.c simply increments said shared variable (a counter of integer type) 250 times after X lines have been read from the text file

provided (which is hp2.txt). Since this is a fixed number of incrementations, and incrementations are a constant time operations – the total asymptotic complexity of the critical section of this code still runs in O(1) time. This means that the wait for the lock will be short, which is favorable for spin locks. T

Example tests:

Spin lock, all thread pool sizes, low contention 2 cores

Spin lock, all thread pool sizes, low contention 4 cores

Spin lock, all thread pool sizes, low contention 8 cores

Spin lock, all thread pool sizes, high contention 2 cores

….

TESTING:

There are two separate testing scripts I have included in this project. One tests all my possible cases of Low/Medium/High core set size, Low/Medium/High thread pool size, and Low/High contention. This script (called testscript) includes 18 tests and was used to look for patterns across all the tests. There is another test script called focusedTests that separates individual tests into categories based on which locking mechanisms is most efficient. This script make it easy to see in which cases you should use each locking mechanism.

RESULTS:

Upon testing all of these cases I noticed there was clearly cases when you should use each locking mechanism. I saw trends that I chose to classify in systems that I dubbed “stressed” or “relaxed”. Stressed systems had a low level of CPU set size and an equal or high level of thread pool size. I called these systems stressed because they will have to do a lot of work scheduling the threads to do work on minimal resources. Relaxed systems had a high level of CPU set size and a strictly lower level of thread pool size. Higher contention levels add to the systems stress.

The first thing I realized was that in a low contention environment – all locking mechanisms were efficient (range of all run-times for tests in low contention: [0,0.2]). However, in cases where the

system was relaxed (high CPUs, low/medium thread pool size), the spin lock and TTASB were on the lower end of the run-time range, and mutex and semaphore would be at the higher end. This is because the spin lock could safely dedicate threads in the smaller thread pool to the large amount of CPUs provided with it's first come first served basis and not have to worry about having aplethora of context switches and other scheduling overhead that comes with with the mutex, and semaphore. Something that was worth noting in these low contention cases was that the TTASB used significantly less CPU power than the other locking mechanisms in a majority of the cases. In some cases TTASB would use 4x less CPU power than the mutex and would run approximately 0.01s slower. So if wait time isn't an important favor, CPU power is expensive, or you want minimal wear on your CPUs then TTASB locks are very practical.

The run-times of the high contention cases provided much more variant data. Here it was very noticeable that in cases of high stress (CPU level ≤ thread pool size level) a pthread mutex or semaphore was a much better choice than the spin or TTASB locks. This is because instead of spin lock's first come first served scheduling being efficient, it is more reasonable to let the pthread_mutex scheduling policy take over (chooses which thread accessed the mutex most recently) to ensure threads don't as easily starved as can happen in spin locks and the TTASB.

Another finding I thought was interesting when testing was that spin locks cause next to context switches, and mutex and semaphores cause a lot. In some cases the ratio of spin lock to mutex voluntary context switches was nearly 500:1, and the involuntary switch ratio of 2:1. So if for some reason the system you are running has exceptionally costly voluntary context switches then it may be more practical to go with a spin lock implementation. These results could have been predicted since spin locks have little business causing context switches and scheduling threads in and out since it keeps a FIFO-esque order of threads being scheduled to free CPUs and only switched out once complete.

CONCLUSION:

From these tests I found instances where each locking mechanism was efficient. In systems of stress (thread pool size level ≥ CPU set size level) the mutex and semaphore were the most efficient options – with semaphore having a slightly quicker average run time (see Figure 1 in Appendix). In systems of low stress or as I defined – relaxed systems (thread pool size level < CPU set size level) – a spin lock becomes the more efficient option (see Figure 2 in Appendix). The only time I found a real use for the TTASB lock was in cases of low contention and lower stress. In these cases, the run-time of the locks vary very little. If a run-time improvement of 0.00-0.02s is negligible in said low stress environment and CPU usage is important to minimize, then it is a great idea to use TTASB as it uses much less CPU power than other implementations in these cases (1-4x less CPU power).

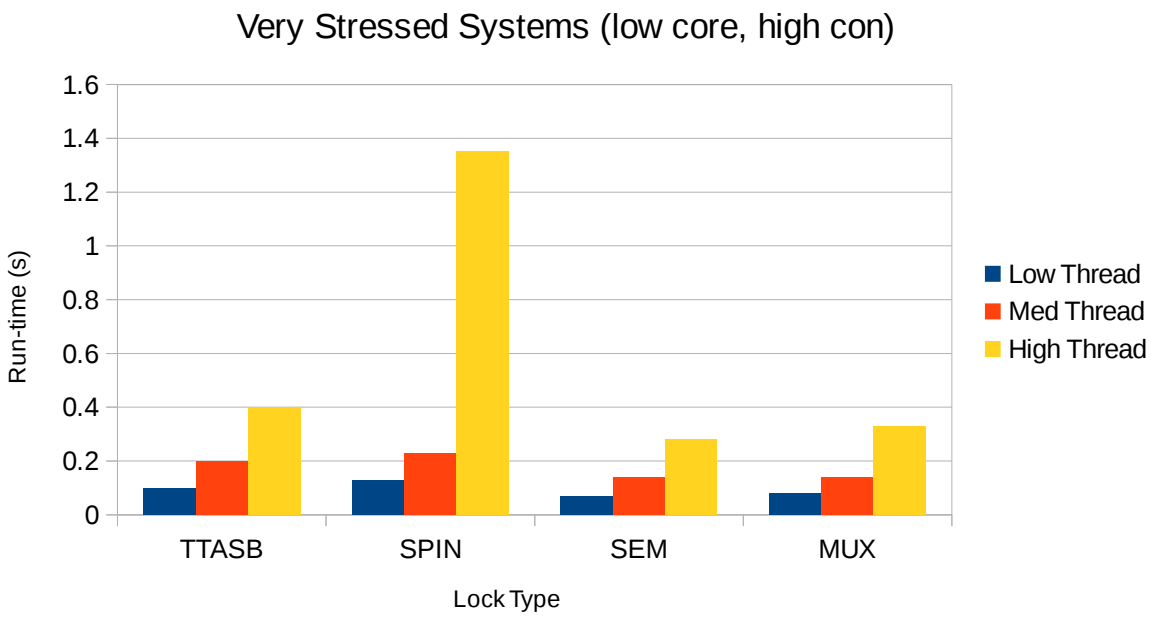
TAKAWAYS:

From this project I learned when to use mutexs, semaphores, spin locks, and test-test-and-set locks with exponential back off. It caused me to view system performance in a broader view after realizing there are multiple factors (CPU set size, thread pool size, contention level) that could cause one locking mechanism to be more efficient that another. While analyzing the data collected I separated systems into stressed (CPU set size <= thread pool size) and relaxed systems (CPU set size > thread pool size) and found out which locking mechanism worked best for each type of system.

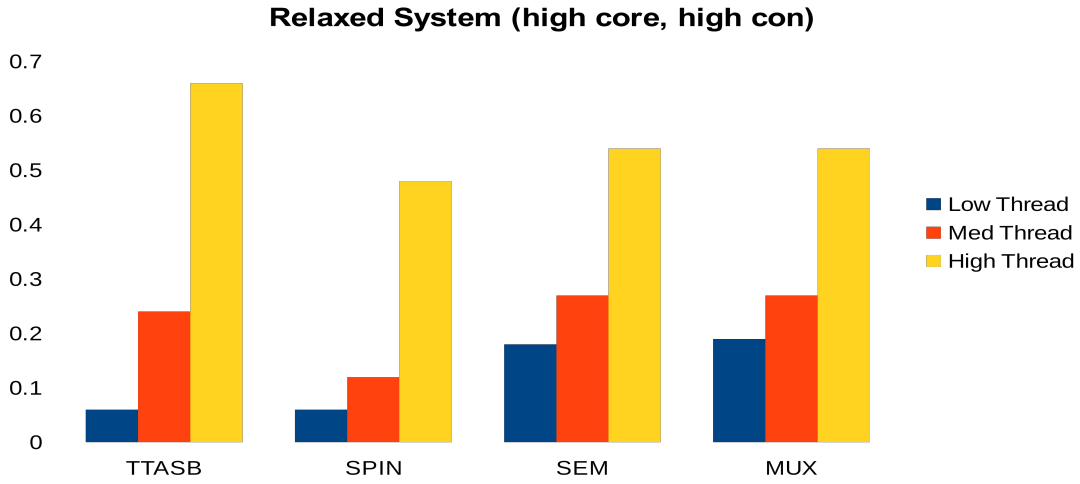
I've now also gained interest and began thinking about the practicality of making a system dynamic as to change locking mechanism based upon changing CPU set size or thread pool size. This could potentially vastly increase the run-time efficiency (use of mutex/semaphore when contention high, and higher system stress) and/or CPU usage (use of TTASB when contention low, and minimal system stress) efficiency in such a system.

APPENDIX

Figure 1



This graph depicts the recorded run-time output in a very stressed system. Here notice how the semaphore and mutex are much more efficient than the TTASB and spin lock. Spin lock gets exponentially worse as system stress load increases.



This graph depicts the recorded run-time output in relaxed systems. Notice how the spin lock is much quicker than the semaphore and mutex in these cases. This is because we have an ample amount of cores to have our threads spin on. However, as stress increases with an increasing thread pool size, semaphore and mutex efficiency begins to approach that of spin lock's.