School of Engineering Science
Simon Fraser University
CMPT 300

Multithreaded Lock Algorithm Analysis

Paul Vu (301169550)

Introduction:

This paper will evaluate various locking algorithms, specifically the Test-And-Set (TAS) spinlock, Test-Test-And-Set (TTAS) spinlock, exponential backoff mutex, and a ticket queue lock. We will discuss their performances as thread size varies in a program, work inside and outside the critical section varies and iterations vary.

Lock Algorithms

The TAS spinlock is the most simple lock to implement of the four locks evaluated. In the algorithm, the lock simply continues to spin on the lock value, checking always to see if the lock is free. If it becomes free, all threads contending for the lock have a chance to obtain the lock however the thread that actually wins the lock is random, thus this algorithm is not a fair algorithm since a thread that just joined can win the lock over a thread which has been waiting for much longer. If the lock is not free, the thread continues to "busy wait" on the lock, continually performing a read and write as TAS is called, thus invalidating the system cache every time.

The TTAS spinlock is similar to the TAS lock, however instead of continually spinning and performing a TAS on the lock, the TTAS only performs a TAS operation when the lock "looks" free. In this implementation, instead of checking the lock-value continuously, the thread will continue checking the lock status to see if it's free or not. As soon as the lock is free, all the threads contending for the lock perform a TAS. Again, this algorithm is random and not-fair since any thread can win the lock.

The backoff lock allows all threads to try to obtain the lock, and if they miss on winning the lock each thread will sleep for a random duration of time. Consequently, every time the thread misses it will sleep for longer durations exponentially. The idea is to ensure we minimize the amount of invalidations which occur when all the threads contend for the lock. This algorithm is also not-fair since if multiple threads awake at the same time the winner will be random. It is also important to note that without a max sleep delay a thread could continually miss on the lock and sleep for a very long time as each miss exponentially grows the sleep time thus inducing thread starvation.

The ticket lock is different from the previous locks as it ensures fairness in lock acquisition. In this algorithm, when a thread attempts to obtain a lock it must first grab a ticket. Each ticket is unique, no thread can have the same ticket number. The thread then compares its ticket number to the one being served. Only the thread with a matching ticket number will obtain the lock, and thus this algorithm is a first in first out algorithm. Unlike the previous algorithms, there is an order in which thread will acquire the lock next.

Results:

In addition to the above algorithms, we also evaluated the performance of the Pthread Mutex and Pthread Spin locks. In comparing all these algorithms, we will evaluate the results when threads, iterations and work inside and outside the critical section is varied. In all graphs provided, the following legend is applied:

Grey: Test-And-Set Lock
Yellow: Test-Test-And-Set Lock
Orange: Pthread Spin Lock
Dark Blue: Pthread Mutex Lock
Light Blue: Exponential Backoff Mutex Lock
Green: Ticket Queue Lock

Unless otherwise specified, the default values for each test are threads = 4, operations outside = 10, operations inside = 10, and iterations = 1,000,000. All tests were performed on a 4 core 8 thread machine.

Thread Quantity

As expected the TAS lock is typically the worst performing algorithm since it consistently spins and invalidates the cache of all the other threads. The TTAS is slightly better in performance since it only performs a TAS when the lock "seems" free, however as soon as it is free all the threads contend for the lock which degrades its performance. In the backoff lock the performance is most optimal, especially as threads scale since the amount of contention is minimal due to each thread sleeping for a random amount of time. In all these locks, priority inversion can impact the results if the scheduler puts the thread inside the critical section to sleep. If this happens, all the other threads will have to wait outside the lock for the thread inside the lock to wake up and complete execution.

In the ticket lock we see that it performs relatively the same as a mutex lock for N > # of hardware threads in a system. This is because since only 8 threads can run concurrently there will be N-8 threads asleep at all times. As soon as the threads increase over the amount of hardware threads in a computer system performance in the ticket lock dramatically increases. In our tests, it took over 7 minutes to complete when thread quantity was 9. The reduction in performance is due to priority inversion, and … Like the previous locks queue locks can suffer priority inversion when the thread inside the critical section falls asleep. To add to this, they can suffer priority inversion in a second way if the thread that is next in the queue has already been put to sleep by the scheduler. During this time, all the threads are waiting for the sleeping thread to wake up and acquire the lock. Even tho the thread is asleep, no other thread can acquire the lock while the thread is asleep due to its FIFO algorithm. Our results are shown in Table 1 and Fig. 1.

Iterations = 1000000 Workoutside	Pthread Mutex	Pthread SpinLock	MySpinLockTAS	MySpinLockTTAS	Mutex Backoff	Queue
1	25	8	13	13	31	12
2	154	24	206	152	66	205
3	183	78	621	232	98	341
4	339	107	829	587	133	475
5	453	126	1368	753	181	613
6	556	208	2045	1071	265	781
7	654	406	2220	1414	389	931
8	761	516	2664	2014	402	1103
9	852	677	3960	2440	682	>420000000
16	1555	1604	9571	6883	1138	>420000000
32	3144	6130	47915	24688	1215	>420000000

Table 1. Lock performance as threads varies

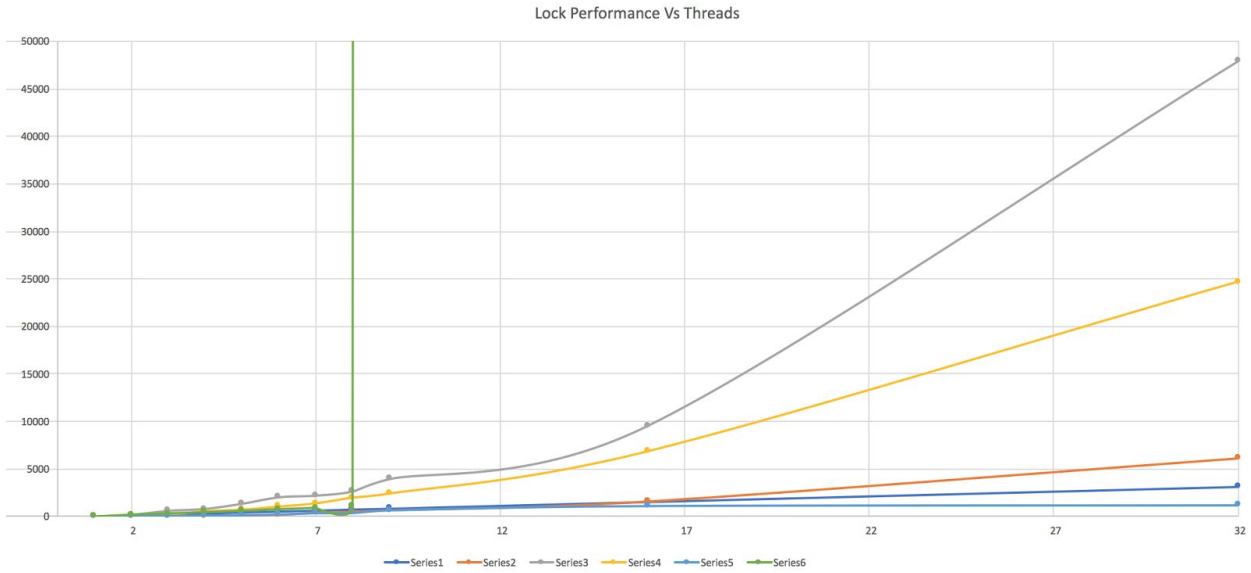

Fig 1. Lock performance Vs Thread Quantity

Work Inside Critical Section

The work inside of a critical section is the serial component of a multi-threaded program. As we increase the amount of work on the inside, we see that the all the locks take longer to execute, this is because we increase the likelihood that the scheduler will put the thread which owns the lock to sleep, thus forcing all other threads to wait until the lock owner wakes up again and can finish execution.
Interestingly, the Mutex lock degrades in performance since the critical section is longer, more threads will miss on the lock and sleep for a longer amount of time. The results are shown below in table 2 and figure 2.

Threads = 4 Workoutside = 10 Iterations = 1000000	Pthread Mutex	Pthread SpinLock	MySpinLockTAS	MySpinLockTTAS	Mutex Backoff	Queue
1	400	272	714	643	144	548
10	507	326	1179	749	230	546
100	1489	890	1779	995	905	1178
500	5601	3966	5807	4011	8955	4306

Table 2. Lock performance as work inside critical section varies

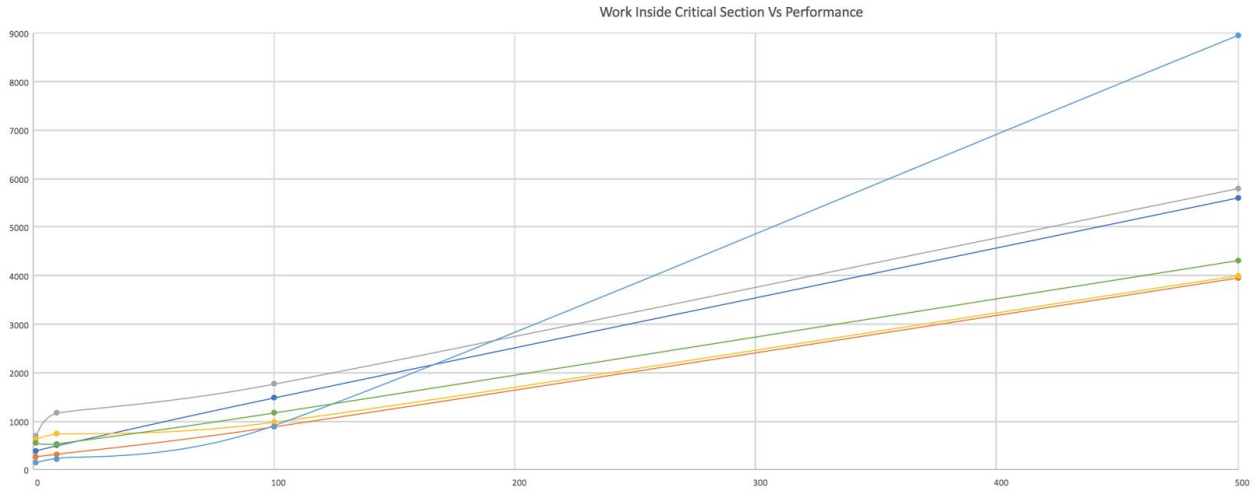

Fig 2. Lock performance Vs Work Inside CS

Work Outside Critical Section

The work performed outside the lock corresponds to the parallel component of a multi-threaded program. In the test we chose to simulate low ratios of work outside to inside keeping work inside constant at 10, we increase work outside up to 500. We do this to see at which point the mutex lock degrades in performance. We can see that with by increasing the parallel to serial component ratio of our program to 7:1 the backoff lock degrades significantly. This is because we increase the likelihood of the backoff lock of missing the lock, thus sleeping for exponentially more time.

Threads = 4 Workinside = 10 Iterations = 1000000	Pthread Mutex	Pthread SpinLock	MySpinLockTAS	MySpinLockTTAS	Mutex Backoff	Queue
1	452	132	1230	674	165	528
10	518	316	982	647	205	560
100	686	496	720	709	897	552
500	1143	1118	1367	1525	2337	1111

Table 3. Lock performance as work outside critical section varies

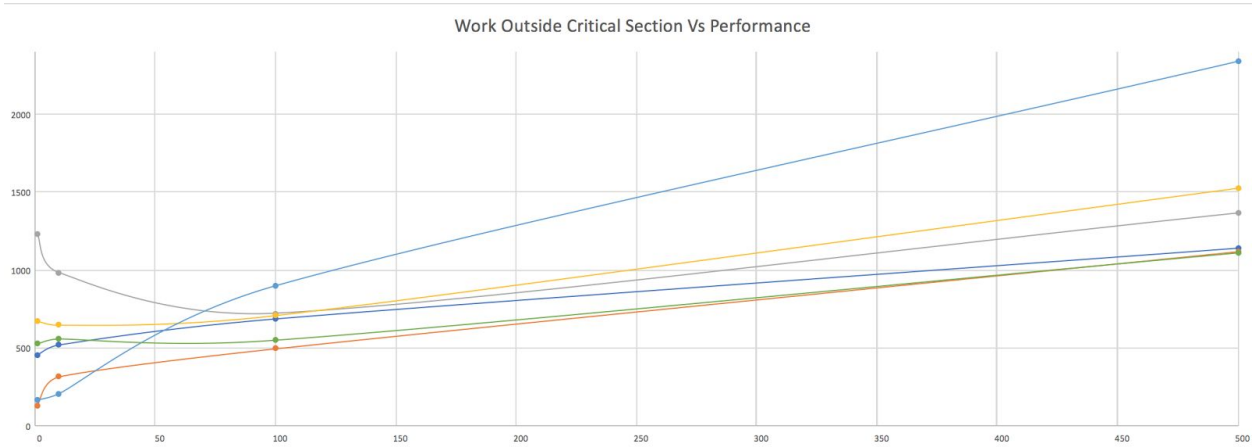

Fig 3. Lock performance Vs Work Outside CS

Lock & Unlock Iterations

Increasing the amount of iterations of acquiring and unlocking the lock simply scales our results linearly. We can see that it is always true that the TAS lock has the worst performance followed by the TTAS lock. The most optimal lock is the backoff lock which is closely related to the performance of the ticket lock. Simply increasing the amount of times each thread attempts to acquire a lock while keeping every other parameter constant will see slower performances since the program is essentially N-times larger with N-times more attempts to acquire the lock. Results are shown in table 4 and figure 4. We can also see that the Pthread mutex lock performs approximately the same as our mutex lock which is to be expected, however it is interesting that the Pthread_spin_lock performs more optimal than the mutex locks! This is likely due to the OS's implementation of the Spinlock, as the iterations increase the OS knows to optimize and reduce the amount of contention on a lock.

Threads = 4 Workinside = 10 WorkInside = 10	Pthread Mutex	Pthread SpinLock	MySpinLockTAS	MySpinLockTTAS	Mutex Backoff	Queue
1	0	0	0	0	0	0
100	0	0	0	0	0	0
100000	60	10	99	73	24	53
1000000	500	302	1381	786	200	540
10000000	5431	3171	11942	7388	5018	5688
100000000	54170	31562	137662	71019	50808	55718

Table 4. Lock performance as lock acquisition iterations varies

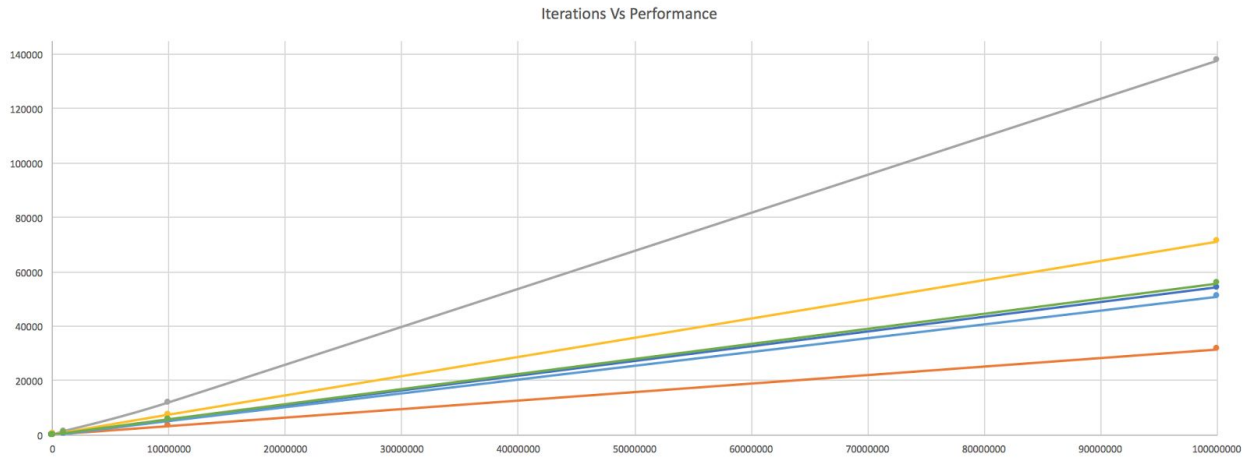

Fig 4. Lock performance Vs Iterations

Conclusion:

In general, we can conclude that the TAS lock has the worst performance due to the constant invalidation of the cache. The TAS lock has slightly better performance since it only performs a TAS operation when the lock seems free, while the mutex lock is consistently the best performing locking algorithm. In all cases these algorithms are non-fair while the queue lock ensures a fair algorithm at the cost of higher likelihood of priority inversion.