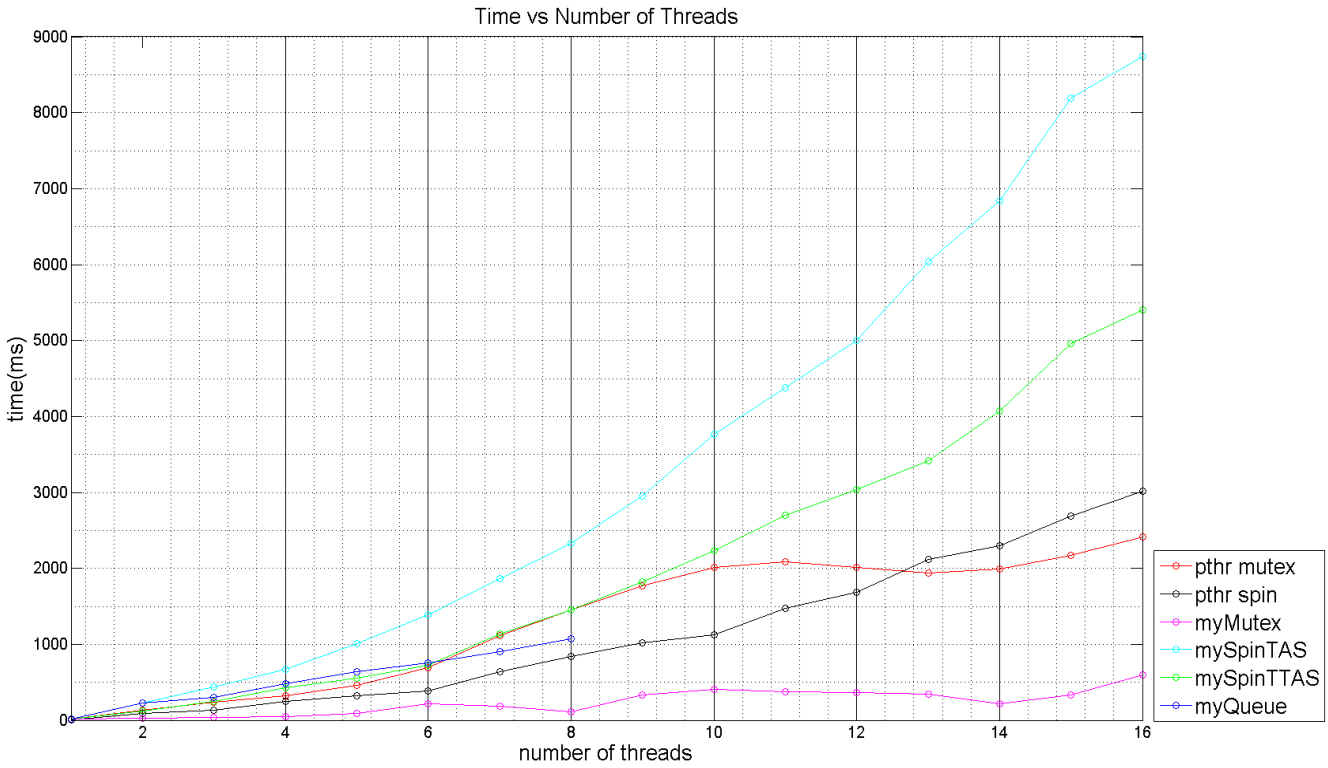
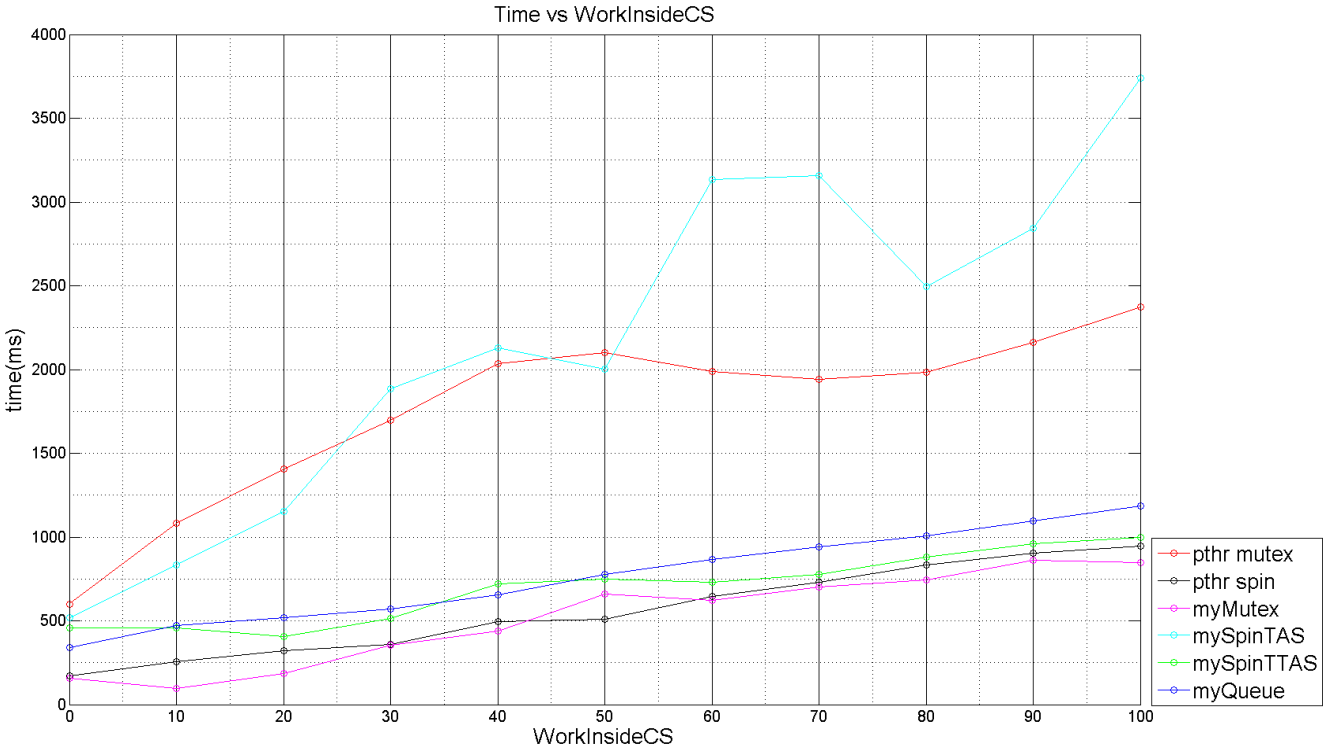
CMPT300 Assignment 3 Report

by Ka Shun (Jason) Chan
ksc19@sfu.ca

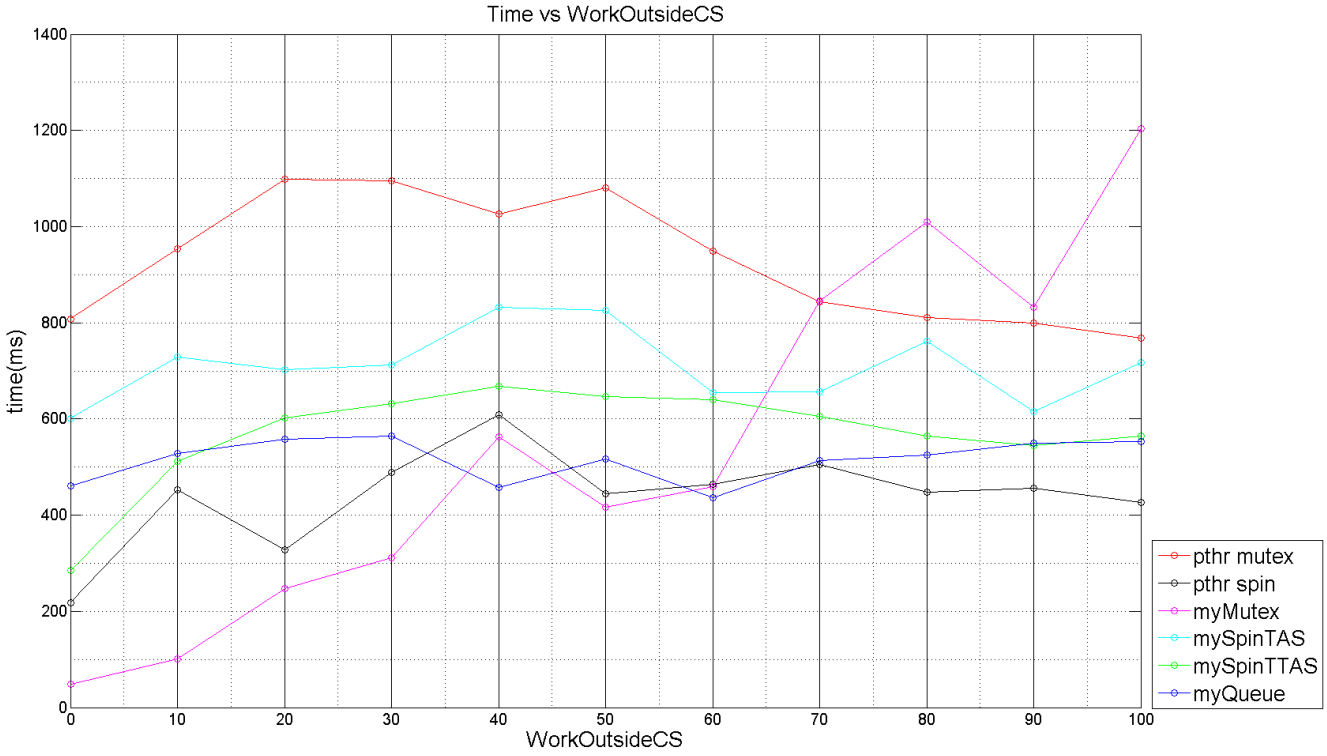
The goal of this report is to examine the efficiency and the implicit relationships between locks and number of threads and work done inside and outside critical section. Besides the Linux built-in pthread mutex and pthread spinlock, in sync.h/.c there are also my own version of spinlocks (using Test-and-Set and Test-and-Test-and-Set), mutex (exponential-back off), and queue lock. All tests are run in SFU CSIL multi-core environment and 1,000,000 iterations unless noted otherwise.



For the first part of analysis, run time is used compared with the increasing number of threads from 1 to 16. Note that the number of threads can run in queue lock is limiting by the number of logical CPU, which is 8 in the SFU CSIL environment. The result shows that spinTAS lock scales up the worst as number of threads increases and myMutex lock performs the best as it does not seem to be affected by the number of threads at all. The bad performance of spinTAS is due to the nature of Test-and-Set method. Test-and-Set tries to acquire the lock by writing on it when it spins to wait and this introduces a huge number of memory bus traffic. On the other hand, spinTTAS looks at the lock bit first without trying to write on it at once. TTAS only writes when the lock “looks” free and this would not cause any bus traffic thus a better performance. In my own implementation of mutex (exponential back-off) lock, instead of constantly contenting like both versions of spin lock, the thread is put asleep to avoid driving up the memory bus traffic thus it performs the best.

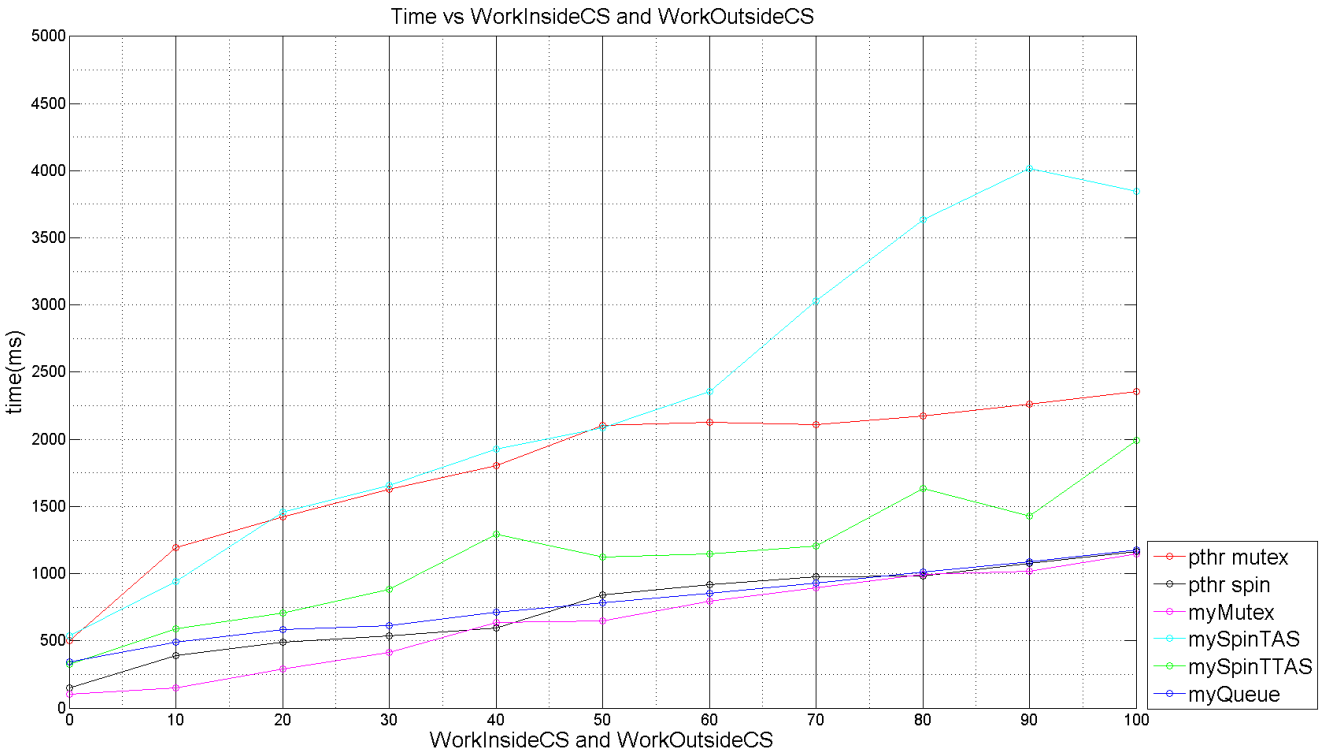


The second analysis is comparing run time with work done inside critical section. An increment of 10 from 0 to 100 for the for-loop in critical section is used to collect the data and all tests are run with 4 threads. SpinTAS performs the worst again due to the unoptimized algorithm and also note that pthread mutex also performs significantly worst than the others.



This is caused by the implementation of pthread_mutex_lock constantly contexting switching into kernel space.

The third analysis is comparing run time with work done outside critical section. An increment of 10 from 0 to 100 for the for-loop outside critical section is used. Interestingly, all locks' run times stay independent of work done outside critical section except for the myMutex lock. This is due to the algorithm of my own implementation of mutex putting the waiting thread to sleep for a larger amount of time than optimal.



The last analysis both work done inside and outside critical section are incremented from 0 to 100 concurrently. As expected, the data shows a similar result to our second analysis.

Over head analysis: this program does not have significant lock overhead as all locks only store 2 to 3 bits of data and critical section only contains one integer variable.