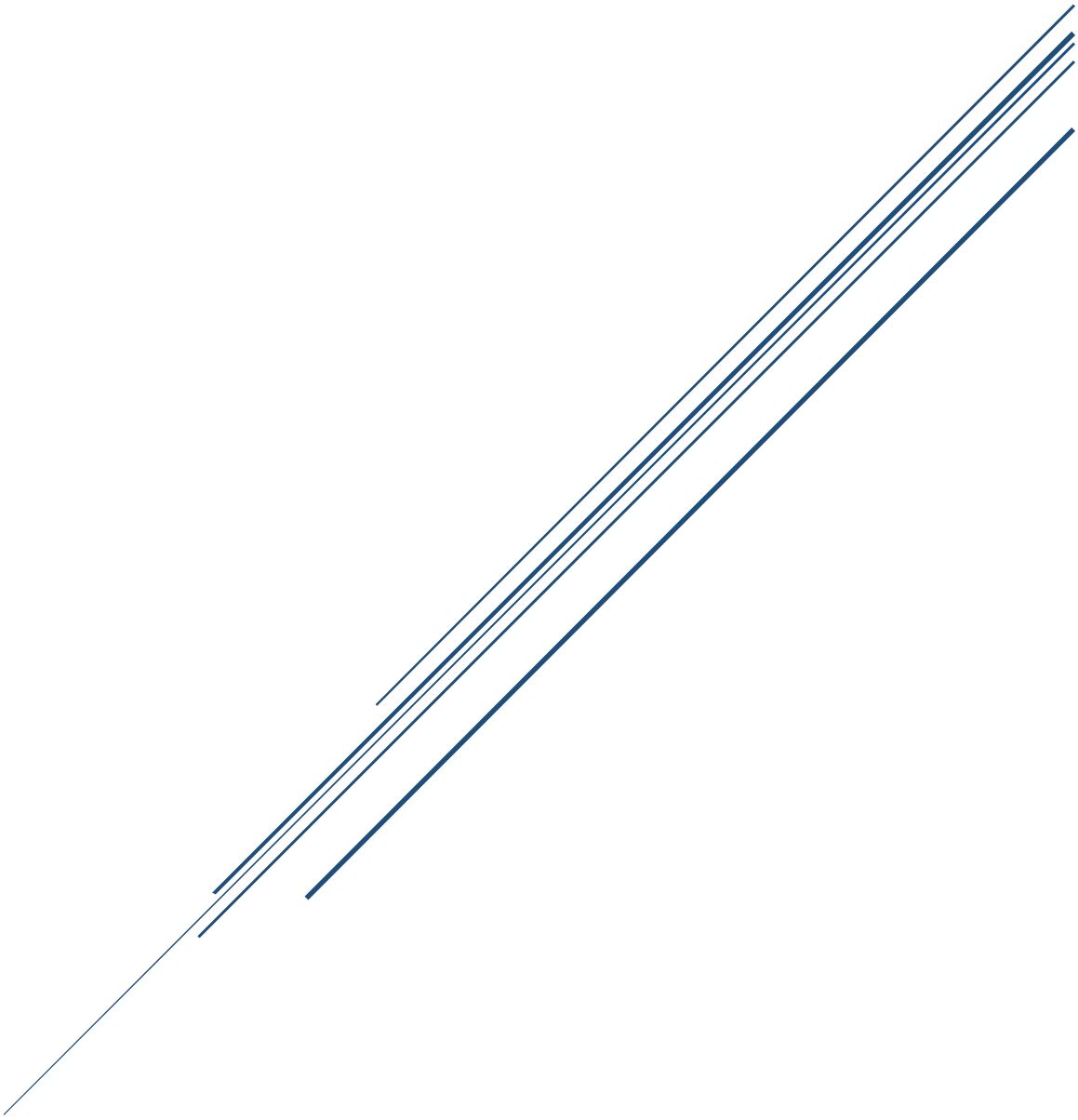
# ASSIGNMENT 3
# SYNCHRONIZATION



CMPT 300
Operating System

Jerry Chienyu Chen
November23/2016

# CMPT 300 SYNCHRONIZATION

**Assignment 03**

| Default Test Points | Time(ms) |
|---|---|
| Pthread mutex time | 896 ms |
| Pthread spinlock time | 253 ms |
| mySpinTAS time | 859 ms |
| mySpinTTAS time | 487 ms |
| myMutexTTAS time | 171 ms |
| myQueueLock time | 533 ms |

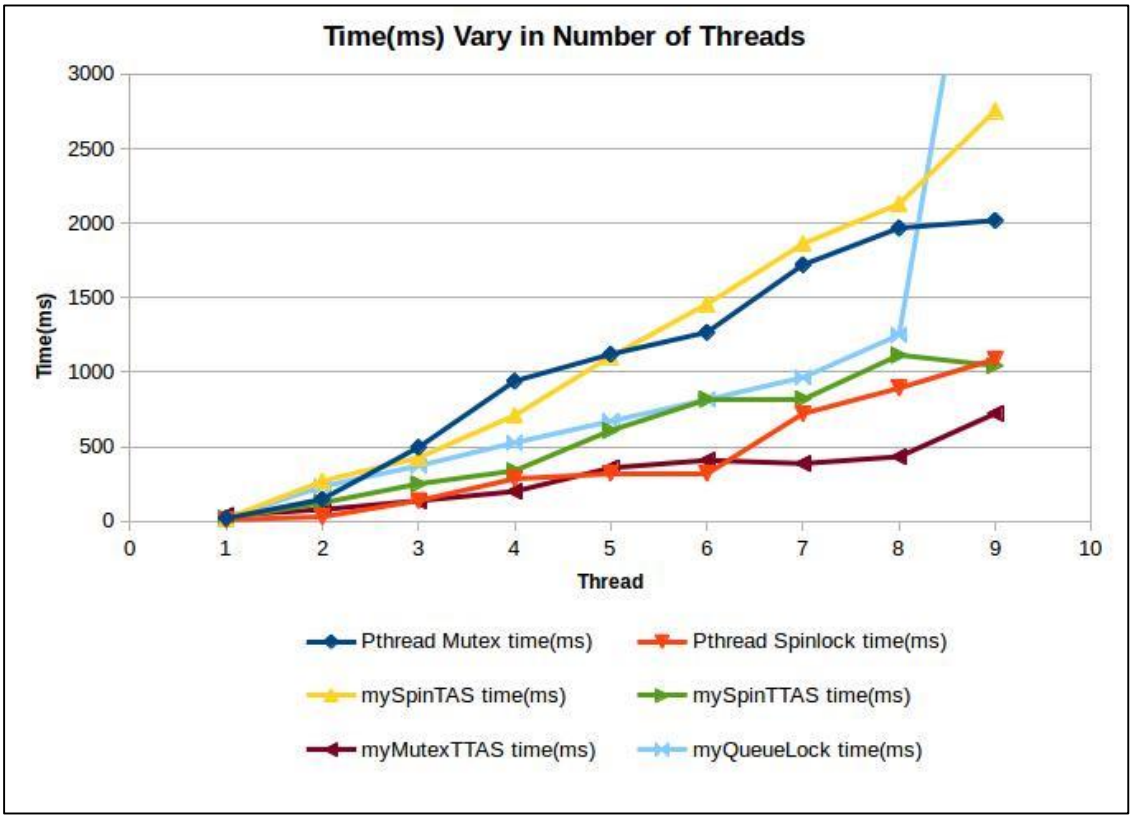
## INTRODUCTION TO THREAD SYNCHRONIZATION

In multi-processor systems, jobs are split into multiple sub-jobs serviced by multiple tasks. Because of this, process and thread synchronization play important roles when multiple processes join up at a certain point after they are done with their tasks to reach an agreement or commit to a certain sequence of action. Speaking of thread synchronization, thread synchronization is a mechanism ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as the critical section. Processes' access to critical section can be controlled using synchronization techniques. Hence, for this assignment, we are asked to implement our own synchronization functionality against two pthread implementations namely the pthread mutex and the pthread spinlock, and also against 3 different locks we are asked to implement ourselves. After all, compare the run time using different synchronization techniques and conclude any interesting observations.

## RESULTS OF VARIATION IN NUMBER OF THREAD

Measuring the time it costs for different synchronization locks on variation in the number of thread, I fixed the number of iteration, workOutsideCS, and workInsideCS as the given default input values just so every thread is doing the same amount of work when the thread is created doing the operations given in its own call back function. Therefore, we can have fair comparisons among the locks when observing the changes in time

with the number of thread increases for different locks.




## Conclusion and Observation of Variation in Number of Thread

After measuring the time it costs for different synchronization locks on variation in number of thread with 1000000 iterations, 0 workOutsideCS, and 1 workInsideCS, we can observe from the figure above that the time of all the locks are linearly increasing when we increase the number of thread over time. This is reasonable because every time a thread is created, it is only incrementing the global c within the critical section iterating workInsideCS many times since it is not doing any work on the variable localCount with 0 workOutsideCS many times outside of the critical section. So when there are more than 1 thread created under this circumstance where only the critical section is doing the work of its call back function when pthead_create is invoked, the more threads there are, the more time it will take on the contention fighting over the locks trying to access the critical section.

As shown on the figure above, we can also observe that the mySpinTAS is taking more time than the mySpinTTAS on average. This is correct because mySpinTTAS checks if the lock is free before the thread actually goes in and tries to grab the lock which prevents the chances of the threads contend over the spin lock. Moreover, TTAS was originally designed to be a better version

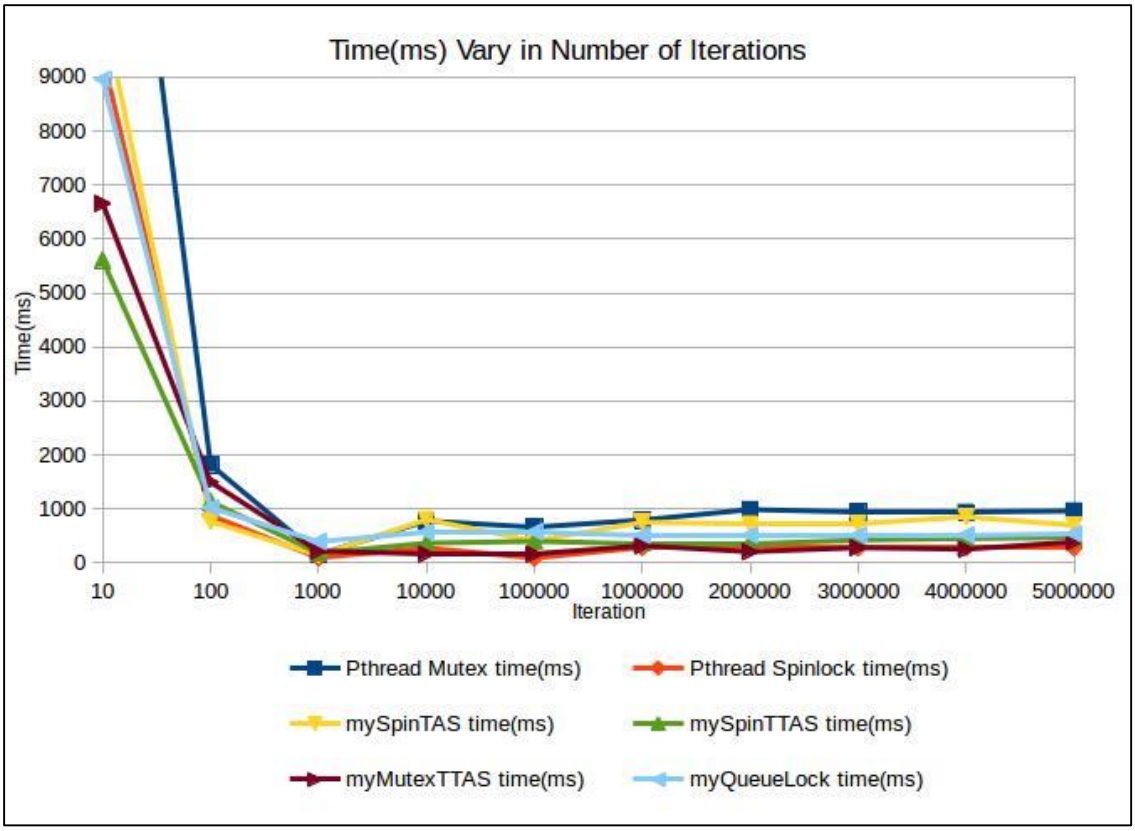
of TAS so theoretically it should have better performance than the TAS method which is being proved practically with the result provided above.

Furthermore, we can also observe that the queue lock has roughly the same performance as mySpinTTAS lock and myMutexTTAS lock is doing the best over time. The reason why myMutexTTAS is doing better than mySpinTTAS is because the spinning method is taking up more time and more overhead than mutex lock method although both of them were implemented with the TTAS method.

Other than the comparisons among the different locks, we can also observe one interesting fact showing on figure above, where the queue lock's time blows up when the number of thread is increased to 9 threads. This interesting fact is related to the physical machine in CSIL. Machines in CSIL are 4 core 8 thread machines. So circumstances where we create only less than or equal to 8 threads, each thread can physically obtain its resource from the physical cores without fighting over the resources. However, once we create more than 8 threads on the CSIL machine, the CPU scheduler is then invoked to schedule the threads since they cannot obtain their own resources equally on the physical CSIL machine. Once the CPU scheduler is invoked, thread switching will then happen at some point to share the resources among overloading threads. And this is the reason why when we create up to 9 threads, the performance time blows up and takes way longer than the performance time of 8 threads or less.

## RESULTS OF VARIATION IN NUMBER OF ITERATION

Measuring the time it costs for different synchronization locks on variation in the number of iteration, I fixed the number of thread, workOutsideCS, and workInsideCS as the given default input values just so every thread is doing the same amount of work when the thread is created doing the operations given in its own call back function. Therefore, we can have fair comparisons among the locks when observing the changes in time with the number of iteration increases for different locks.

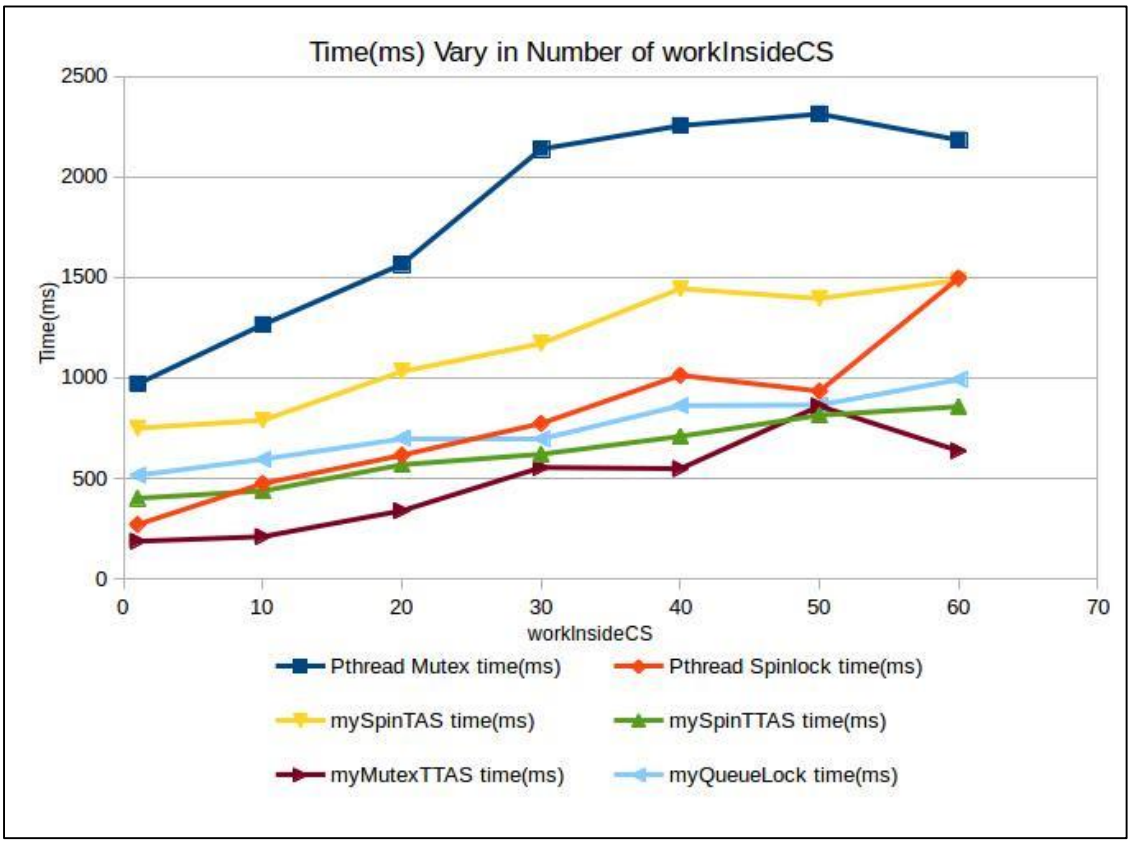



## Conclusion and Observation of Variation in Number of Iteration

After measuring the time it costs for different synchronization locks on variation in number of iteration with 4 threads, 0 workOutsideCS, and 1 workInsideCS, we can observe from the figure above that the performance time of all the locks are decreasing over time rapidly for the first 1000 iteration change and then soon stabilized after the 1000 iteration. The reason behind this is that when the threads are created with small iterations, for instance 10 iterations, of work being done in the call back function in pthread_create, there are not as much work done so not many of them are cached. In another words, the cache is not fully utilized before the work iteration finishes so the performance time is not efficiently great for low iterations. Whereas when the threads are created with large iterations, for instance 1000000 iterations, of work being done in the call back function in pthread_create, there are a lot more work done. Therefore, the cache is fully utilized before the iteration finishes which the later iterations of large iterations have better performance time using the cache. After all, large iterations have better and more stable amortized performance time.

# RESULTS OF VARIATION IN NUMBER OF WORKINSIDECS

Measuring the time it costs for different synchronization locks on variation in the number of workInsideCS, I fixed the number of thread, iteration, and workOutsideCS as the given default input values just so every thread is doing the same amount of work when the thread is created doing the operations given in its own call back function. Therefore, we can have fair comparisons among the locks when observing the changes in time with the number of workInsideCS increases for different locks.



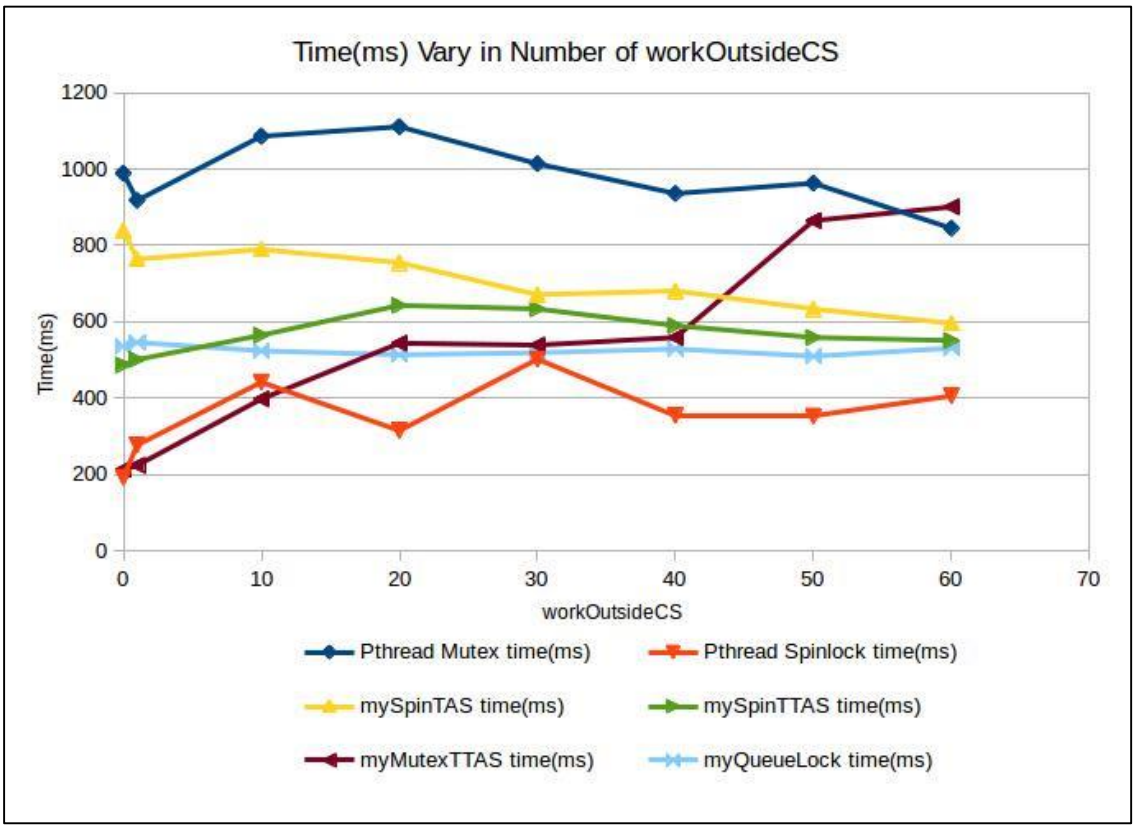

## Conclusion and Observation of Variation in Number of workInsideCS

After measuring the time it costs for different synchronization locks on variation in number of workInsideCS with 4 threads, 1000000 iterations, and 0 workOutsideCS, we can observe from the figure above that the performance time of all the locks are linearly increasing over time because increasing workInsideCS over time is just adding more work to the critical section that a thread has to do. One interesting fact about this is where the pthread spinlock starts off with great performance better than mySpinTTAS lock and the queue lock when workInsideCS is small. But once the workInsideCS increases up to 20, its performance becomes roughly the same as the performance of mySpinTTAS and the queue lock. And it eventually has worse performance time than both mySpinTTAS and the

queue lock when the value of workInsideCS reaches 60. Therefore, we can conclude that pthread spinlock is better to be used when the amount of work need to be done in the critical section is small. Comparing with the pthread spinlock, both mySpinTTAS lock and the queue lock are better choices than the pthread spinlock when the amount of work need to be done in the cirtical section gets greater. Other than that, the myMutexTTAS lock is doing the best among all locks over time, so it would be the best choice of lock to use overall.

## RESULTS OF VARIATION IN NUMBER OF WORKOUTSIDECS

Measuring the time it costs for different synchronization locks on variation in the number of workOutsideCS, I fixed the number of thread, iteration, and workInsideCS as the given default input values just so every thread is doing the same amount of work when the thread is created doing the operations given in its own call back function. Therefore, we can have fair comparisons among the locks when observing the changes in time with the number of workOutsideCS increases for different locks.



## Conclusion and Observation of Variation in Number of workOutsideCS

After measuring the time it costs for different synchronization locks on variation in number of workOutsideCS with 4 threads, 1000000 iterations, and 1 workInsideCS, we can observe from the figure above that the performance time of the locks are not linearly increasing anymore, this is because every time a thread is created, it spends some time to increment the localCount before it goes and grab the lock trying to enter the critical section and start incrementing global c. Therefore, as more and more threads are created, they do not contend over the locks to enter the critical section as fiercely as when the amount of work done outside of the critical section is 0. Since the contention is not as dramatic anymore, the amount of performance time wasted at the contention is not as much anymore, so the overall performance time is barely increasing as the number of workOutsideCS increases. As what I have tested, increasing the number of workOutsideCS also introduces fairness to all the threads created where each thread gets chances to obtain the lock to run the critical section instead of one single thread dominating the critical section once every long time.

# TESTING ENVIRONMENT