An Analysis of Locking Techniques in Multi-Threaded Programming

Shawn Anderson
Ka Hang Jacky Lok

I. INTRODUCTION

The purpose of this report is to explore the implementation and performance of three mutual exclusion (MutEx) locking techniques. MutEx is a critical paradigm in multi-threaded computing. It allows multiple threads to work with shared data. It ensures that critical sections of code, such as updating a shared variable, are not executed by multiple threads simultaneously, which has the potential to produce undesired behaviour. The three locking techniques implemented are: a basic spinlock, an exponential backoff lock, and a queue lock.

II. BACKGROUND

A. Lock Basics

In the context of this paper, locks are data structures which are used to maintain knowledge about critical sections of code. Knowledge which allows threads to work synchronously on shared data, while maintaining correctness of tasks. Locks secure Critical Sections (CS) of code, in which threads operate on shared data. Locks are implemented using atomic operations. Atomic operations are operations that are executed transactionally in the sense that they are completely executed, or not executed at all, ie. they do not allow interleaving before completion. If a lock serves threads in the order that they have arrived, it is said to be a First In First Out (FIFO) lock. When threads are waiting for a lock, we refer to them as 'spinning', in the context of this paper, spinning does not progress a task, it is a form of waiting for a lock. In other words, busy waiting is not implemented in the scope of our experiments.

Deadlock is a situation where there exists a cycle of threads, all waiting on each other to relinquish control of a lock, resulting in all threads coming to a complete halt and not making any progress. If a lock is not recursive, then a deadlock will occur if a thread holding a lock attempts to grab it's own lock.

B. Spinlocks

The most simple type of lock is the appropriately named Test-and-Set(TAS) lock. In the case of a TAS lock, we instantiate a data structure, which we call my_splinlock_struct to represent the lock. This data structure has a lock attribute which will always be either 0 or 1. In the TAS method, if lock is set to 0, then the critical section is free to be entered, if the lock is 1, then the lock is held by another thread, thus we must spin (keep checking the lock). Once the holder of the lock has exited the critical section, it will set the lock back to zero, thus the next thread to check the lock will be able to obtain it and enter the critical section.

A key component of the spinlock is an atomic test-and-set(tas) operation. A tas operation allows a thread to check the value of the lock, and, if the lock is free, lock the lock, all in one atomic operation. This operation is atomic, meaning it either completes or does not, it can not be interleaved with any other operations. This tas operation is essential for the synchronization of the TAS method, it ensures that two threads do not both check and lock the lock at the same time.

There is a cost for using atomic operations. They use excessive clock cycles, compared to their non-atomic analogues, due to their increased complexity. However, there is a more profound issue that must be addressed with the tas operation in the TAS method. This issue arises from the concept of cache coherence, which reduces bus access to memory by accumulating data changes in the cache, only accessing memory when necessary.

If threads waiting for the lock are all spinning on checking the lock, then they are invalidating cache lines. Spinners are missing the cache, thus going to the bus. When a thread goes to release a lock it will be delayed behind spinners. These issues result in excessive bus bandwidth by spinning threads and increased release/acquire latency for the lock. This problem is solved by the implementation of the Test-and-Test-and-Set (TTAS) lock.

The TTAS lock is nearly identical to TAS, with an additional 'Lurking' stage. In the lurking stage, the thread spins on a simple read of the lock rather than a tas operation on the lock. The key concept here is that a simple read does not invalidate cache lines. A lurking thread will check if a lock 'looks' free. When the lock looks free, the thread will enter the 'Pouncing' stage, in which it calls tas to acquire the lock, if tas fails, it returns to lurking phase. The pouncing stage is identical to the TAS implementation, thus TTAS is simply TAS, wrapped in an additional loop which reads the lock.

In practice, TTAS performs much better than TAS, especially as the number of threads grows. However, it has it's own issue related to cache coherence. When the lock is released, all lurking threads will pounce on the lock that now appears free. This creates an 'invalidation storm' in which every lurking thread sequentially reads that the lock is free, and then performs a tas operation, invalidating the other threads caches. This invalidation storm problem is resolved by the Exponential Backoff Lock.

Simon Fraser University

C. Exponential Backoff Locks

The Exponential Backoff Lock solves the cache coherency invalidation issues with a simple and elegant solution. Start with the TTAS implementation, but add in the concept of backoff. That is, spin until the lock appears free. If the lock appears free, try to grab the lock with tas. If the tas fails, then it is obvious that there is high contention for the lock. In this case, go to sleep for a random amount of time. This randomness essentially enforces an ordering of the threads competing for the lock, since each thread will go to sleep for a different amount of time. Thus we have eliminated the invalidation storm problem.

Our implementation is exponential, that is, each time a thread consecutively finds that a lock is under hi contention, it will double the amount of time that it sleeps for. By doing this, lock attempts become more and more spaced out, resulting in minimal cache invalidations. Exponential Backoff Lock has the highest performance in our experiments, even outperforming the pthread mutext lock implementation.

D. Queue Locks

There are multiple types of queue locks, the one that we are experimenting is Ticket Lock[1]. Our implementation is inspired by the pseudo code presented on the wikipidia page. However, our implementation uses the Compare-and-Swap (cas) atomic operation instead of the Fetch and Inc atomic operation. Our ticket lock works exactly as a ticket system when you go to a deli, or to SFU fincancial services. The lock maintains two counters: now-serving, and next-ticket. Next-ticket is always equal or greater to now-serving. When a thread makes a request to access the critical section, it uses the cas atomic operation to take a ticket, and increment the next-ticket attribute, all in an atomic operation. The atomicity ensure that no two threads get the same ticket number, and that all valid tickets are held by some thread. A thread waiting on the critical section will spin until the now-serving attribute of the lock is equal to the ticket number that the thread possesses.

Queue lock is not the most efficient of the locks. It does however, posses some potentially desirable qualities. A queue lock guarantees FIFO, which introduces a notion of fairness to acquiring the lock. FIFO also eliminates the possibility of starvation, which is when some thread is never able to acquire the lock.

E. Recursively Aware Locks

Our lock implementations are recursively aware. This means that if a thread attempts to lock a lock that it is already holding, it will not deadlock itself. Instead, it will require and additional unlock to finally release the lock. This is implemented by adding a lock_owner attribute to our lock data structure, as well as changing the binary lock attribute to an integer counter attribute. So a lock is aware of its current holder, if that holder attempts to lock the lock again, the counter is increased. When a thread unlocks the lock, the counter is decreased. When the counter goes to zero, the lock is released, thus a thread must unlock as many times as it has locked to release the lock.‘‘

III. Evaluation

To evaluate the performance of the locks, a test is devised. In this test there exists a counter which is initialized to zero. Each thread increments this counter a set number of times. Threads are running concurrently, thus the process of incrementing the counter becomes a critical section. Incrementing a counter is a critical section because increment is not an atomic operation. If this test is run with no locks around the increment, than an incorrect final count is produced. If a faulty lock is used to secure the critical section, than a an incorrect final count is produced. Thus correctness was the first goal of implementation.

Once locks where shown to be implemented correctly, we were able to run our test with varying parameters to evaluate the performance of our locks in different scenarios. These parameters include: number of threads, amount of work done in the critical section, and amount of work done outside of the critical section.

A. Experiment 1: Thread Numbers

In this experiment we test how the locks perform as the number of threads increases. We hold amount of work inside CS, and amount of work outside CS both constant at 300. Number of threads is increased exponentially in each trial.

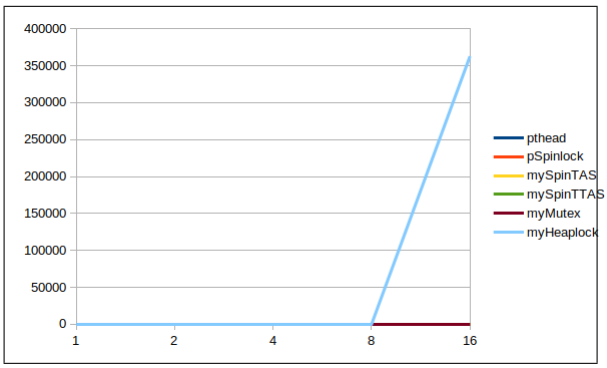

Fig. 1. Thread Test: Number of Threads vs Time(ms)

During the experiment, we found that one of the locks takes exponentially more time than the others when increased the threads, as seen in Fig. 1. Figure 2. shows a graph without heap lock's time at thread 16, and it is obvious that most of the lock just works as expected except heap lock. We are curious about the reasons and conditions that lead to this problems, so we have done varies of tests against heap lock, such as, increase the threads from 1 to 11, increasing other parameters, checking the implementations. And we found that this would only happen when the threads are more than eight, which is the hardware threads of the lab computer! After research, we found that the cause of this is "Priority inversion". Each hardware thread has its only priority, but in heap lock, we give each thread a ticket and let them wait until their round, which may eventually put the higher priority task to the queue dues causing the priority inversion problem.

This problem could be solved if we yield and calling thread to relinquish the CPU when the job is in the queue, which can be done by calling SCHED_YIELD on the while loop.

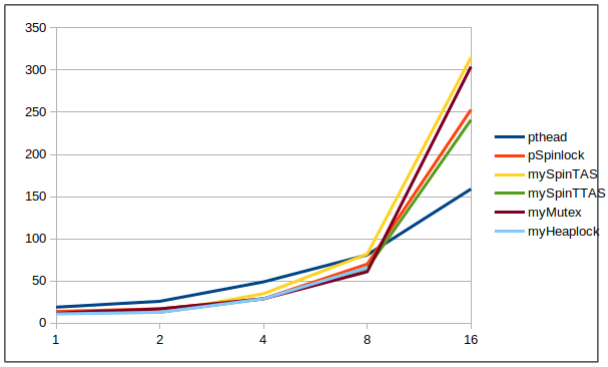

Fig. 2. Thread Test: Number of Threads vs Time(ms) excluded heaplock

B. Experiment 2: Outer Loops

In this experiment we test how the locks perform as the number of work done outside the critical section or the outer loop in short. We hold amount of work inside CS constant at 300 and number of threads at 8.

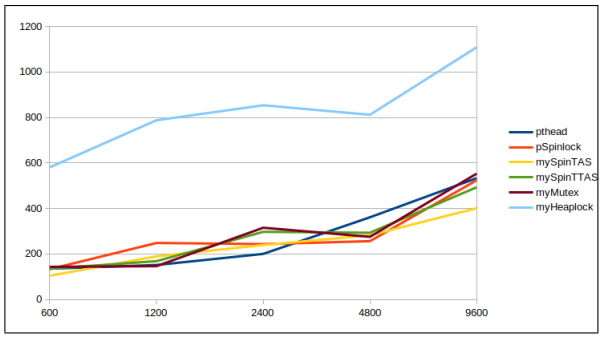

Fig. 3. Outer Loop Test: Number of work done outside critical section vs Time(ms)

The graph shows in average, heap lock takes more time than other lock when work is done outside the critical section increase. However, the figure also matches our assumption that increasing the work outside the critical section would not consume much time since the job could be done in parallel.l.

C. Experiment 3: Inner Loops

In this experiment we test how the locks perform as the number of work done inside the critical section or the inner loop in short. We hold amount of work outside CS constant at 300 and number of threads at 8.

We think that increasing the work done inside the critical section would increase the work time substantially. The among of time used in Inner Loop is sequential, meaning that each thread would need to wait for other threads to finish that amount of work. Figure 4 shows us that it takes way more time than increasing the outer loop.

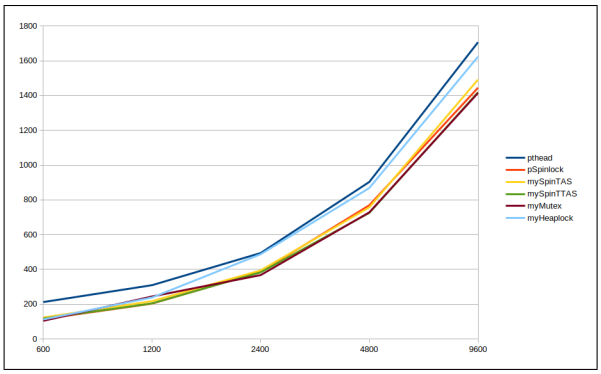

Fig. 4. Inner Loop Test: Number of work done inside critical section vs Time(ms)

D. Experiment 4: Iterations

In this experiment we test how the locks perform as the number of iteration increases. We hold amount of work inside CS, and amount of work outside CS both constant at 300 and number of threads at 8.

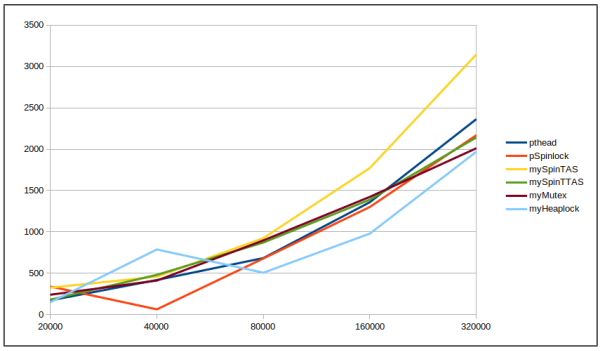

Fig. 5. Thread Test: Number of Iterations vs Time(ms)

Increasing iterations will increase the number of lock cycles hence increase even more time. Figure 5 shows the time used when iterations increases. The average time use is more than outer loops, therefore proved our assumption.

IV. CONCLUSIONS

When we are dealing with multi-threading, we need to be aware of many things, such as lock efficiency, deadlock, priority, and hardware thread, and each of them may lead to some unexpected behavior. We think that knowing more about hardware operating cycle would help us to excel multi-threading. And we would continue to explore more technique about locks in the future.

ACKNOWLEDGMENT

Thank you to the instructor Ryan Shea for clearly defining these concepts in lab and lecture, as well as producing high quality slided which document these concepts.

REFERENCES

[1] https://en.wikipedia.org/wiki/Ticket_lock
[2] https://sfucloud.ca/cmpt-756/wp-content/uploads/2018/01/multi-core.pdf