Locking/Synchronization Primitives

Lynus Vaz, Rakesh Mahadasa and Mugilan Mariappan

CMPT 740 – Project report

1. Introduction

Locks are synchronization mechanisms used in systems to protect objects shared between multiple threads running in parallel. In the case of database systems, these shared objects can be resources like tables, rows, cached variables etc. Locking these shared objects prevents inconsistencies and provides guarantee with respect to the data. But the design, choice and implementation of locks has a large impact on the performance of the system, and can result in problems like starvation, cache line contention, etc. Modern multi-socket architectures and large NUMA systems already experience latency problems due to remote data access. In addition to this, poor lock design can result in additional overhead. In this report, we will discuss three widely used locks: spin lock, ticket lock and MCS lock and compare their design, performance and drawbacks.

2. Experimental Setup

We ran our experiments on a google cloud machine with the following configuration:

S.No.	Specification	Value
1.	Number of NUMA nodes	2
2.	Number of physical cores per node	24
3.	Hardware threads	Yes
4.	Processor speed	2GHz
5.	RAM	84GB

We allocated the variables used in the lock and unlock paths on their own cache lines (aligned to 64 bytes) to avoid false sharing.

The Intel instruction set reference recommends using the PAUSE assembly instruction in all spin loops to prevent a memory order violation and improve performance[4]. Accordingly, we used the same in the spin loop for all our lock implementations.

2.1 Benchmark

In total, the threads acquire and release the lock 'K' times. This workload is divided equally between 'n' threads, so that each thread acquires and releases the lock (K/n) times. For our testing, we set K to 4194304. The critical section is a dummy loop of m iterations. Each benchmark is run for 2, 4, 8, 16, 24, 32, 48 and 96 threads, for each lock implementation. All the final values mentioned in the paper are the average values obtained across 5 independent

runs of the benchmark. For low critical section workload, we execute a busy loop for 128 iterations. This is the critical section specification used in all our experiments unless specified otherwise.

3. Evaluation

3.1 Spin Lock

We first implemented a dumb spinlock that spins until an atomic compare-and-swap on the lock variable succeeds.

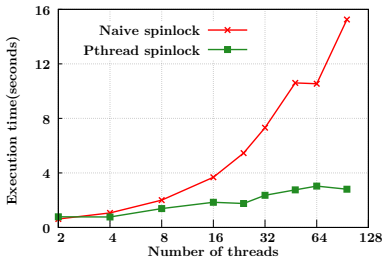

Figure 1a: Naive spinlock vs Pthread spinlock

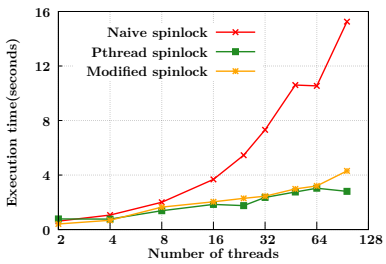

Figure 1b: Modified spinlock

As seen in Figure 1a, we observed that our naïve spinlock implementation performed very poorly compared to the pthread spinlock implementation. We observed that this is due to the overheads incurred in compare-and-swap (CAS) operation. So, we modified our spinlock implementation by performing atomic fetch to read the current status of the lock variable. We attempt CAS operation only when the status is unlocked. By this, we replace most of the heavy-weight CAS instructions into light-weight fetch operations. This gives us performance similar to pthread spin lock implementation as shown in Figure 1b.

Spin lock contention evaluation

In spin lock, every thread keeps checking the same lock variable. To reduce the contention, we varied the backoff time(i.e the time a thread backs off before checking the lock variable again) and measured the impact in Figure 2a. The backoff time is simulated by a “x” number of dummy instructions. We ran the experiment with no backoff time, constant backoff time (500 instructions), random backoff time (0-1000 instructions). We received slight improvements while using a backoff time and random backoff time performed the best among the three.

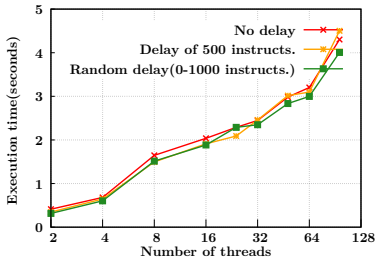

Figure 2a: Spinlock with varied backoff delays

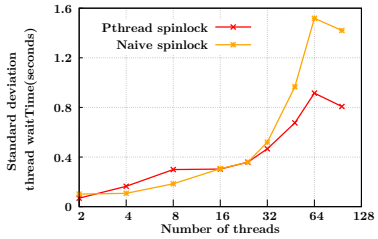

Figure 2b: Cumulative wait times for Pthread spinlocks and Naive spinlocks

One major drawback of spinlocks is that there is no guaranteed order by which threads acquire the locks. To evaluate this, we measured the cumulative wait times incurred by each thread. As seen in Figure 2b, there is a high variance in the cumulative wait times of different threads indicating that some threads do indeed suffer from starvation.

3.2 Ticket Lock

To ensure fairness in lock allocation, we then implemented ticket lock. A ticket lock consists of two integers: *current* and *last*. Each thread obtains its *ticket* by atomically fetching and incrementing the *last* variable. It then loops until the *current* variable becomes equal to its *ticket* and it can enter the critical section. Releasing the lock consists of incrementing the current variable.

Thus, the ticket lock enforces ordering between lock requests. However, this still suffers from the problem of Cache-line bouncing.

Cache-line Bouncing

Modern CPU cores use a layer of cache levels to communicate with memory. Executing CPU instructions on data available in cache is much faster than retrieving it from RAM every time the data is needed. The cache works on a cache-line size of typically 32-128 bytes, and is 64 bytes on the x86-64 platform. The cache checks if the data is already present in its memory, if not it will read a line of data into the cache from the memory. The system should guarantee that the copy of data present in the cache line is same as the corresponding memory.

When a CPU modifies data present in its cache, the cache marks the line as modified. But instead of writing data immediately, the cache writes the dirty lines to memory in batches in order to benefit from multiple writes to the same cache line. All the lines with the same data should be invalidated so that stale values are not used by other cores. This incurs a high communication cost. We have cache coherence protocols to maintain consistency with multiple caches lines, which we are not going to discuss in this report.

When a core completes its critical section and marks the lock as free, the cache line containing that data will be invalidated in all the cores. All the cores must read the status of the lock again from memory.

3.3 MCS Lock

To prevent the problems of starvation and cache-line bouncing, we then implement the MCS lock[1]. Here, each core spins on a local variable to eliminate cache line bouncing. When a thread tries to acquire the lock, it provides its own lock structure. The next field of the main lock is atomically set to its own local structure. The atomic operation returns the old next pointer in the MCS lock, that is the pointer to the structure owned by the last waiting thread (end of the queue). If it returns NULL, there is no contention for the lock: the thread acquires the lock and returns. But when a second thread tries to acquire the lock, it gets a non-NULL value. This implies the lock was already acquired by a different thread. It stores its own structure address in this lock's next pointer. It then spins on its local locked variable.

When the thread currently holding the lock releases it, it atomically does a CAS with the expected value as its local structure and new value NULL. If this succeeds, it means that no thread is waiting for the lock. But if there are threads waiting in queue, then it changes the locked variable in the structure of the next thread in the queue.

The next thread waiting on its local variable observes the locked variable is updated by its predecessor, and it enters the critical section. This guarantees that the locks are acquired in order and cache bouncing is eliminated.

The execution times for all the different types of locks is shown in Figure 3. We found that ticket lock takes more time than the modified spin lock (explained in section 3.1). We suspect that the OS regularly schedules out some of the threads waiting on the locks. This in addition to the FIFO ordering ensured by ticket lock could possibly result in other waiting threads in the CPU to wait for the thread which was scheduled out but was earlier in the queue. This also happens for MCS locks, but the performance benefits of preventing cache-line bouncing offsets the impact of the OS scheduler. As the number of threads increase, the benefits of MCS lock become apparent compared to the other spin locks and hence MCS locks are more scalable than ticket locks because of its superior cache benefits.

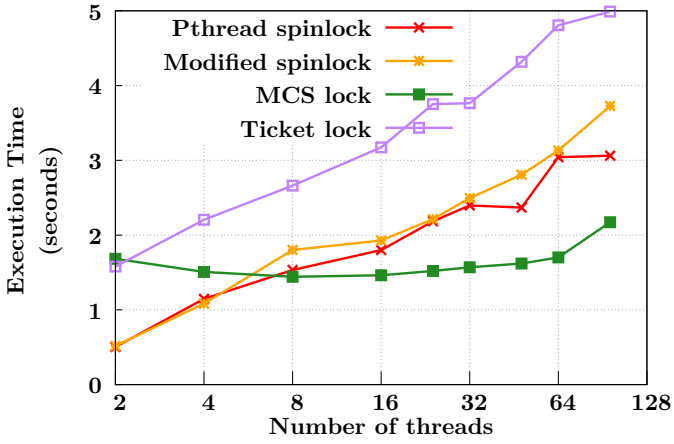

Figure 3: Execution time for spinlock implementations

We also increased the amount of work done in the critical section (busy loop of 4096 iterations) and repeated the experiment.

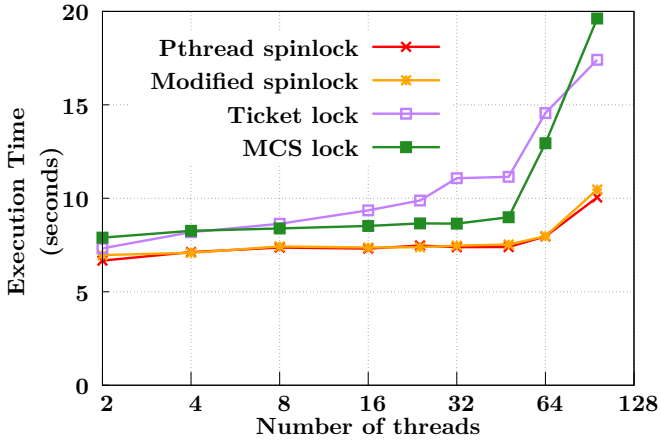

Figure 4: Execution times for spinlock implementations with larger critical section

As seen in Figure 4, the ticket lock and MCS lock have higher execution times than pthread spinlock and the modified spinlock. We believe this is because the OS schedules out threads due to the larger critical section. The FIFO ordered locks are affected more since if the next thread in queue is scheduled out, no other thread can acquire the lock until it is scheduled back in. The pthread spinlock and modified spinlock do not require threads to acquire the lock in order. Therefore, any blocked thread can acquire the lock once it is released by a thread that completes the critical section.

We also observed the cumulative time each thread waits on a lock for different locks as shown in Figure 5a and 5b. Ticket locks and MCS locks have negligible deviation in their wait times as they ensure ordering while granting lock requests.

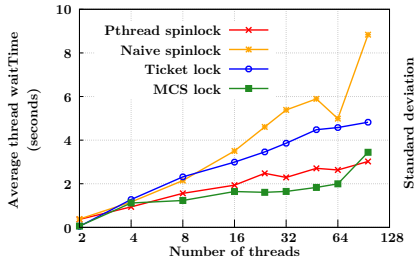

Figure 5a: Average wait time for spinlock implementations

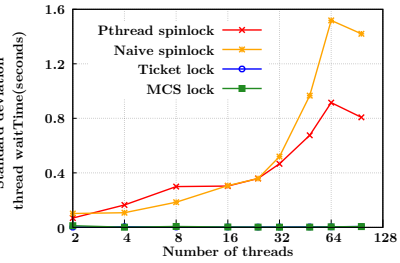

Figure 5b: Standard deviation for spinlock implementations

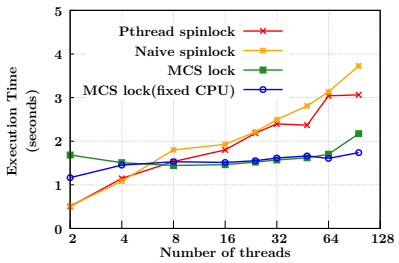

Figure 5c: Execution time when locking thread to a CPU

Finally, we ran an experiment to see if any performance gain can be achieved by locking a thread to a CPU core. As seen in Figure 5c, we did not gain any appreciable performance gain through this.

Implications of MCS lock

In real world applications not all lock users are created equal [2]. Infrequent code paths which request MCS Lock doesn't need fair treatment like frequent code paths. It is also observed that frequent code paths for spin lock are simple while infrequent ones are complex. So, by forcing complex code paths to send the context to acquire MCS lock will reduce programmability. To facilitate easy refactoring of code, we can decouple the frequent and infrequent code paths where simple frequent code paths use MCS lock and the rest use general spin lock. This will lead to starvation for infrequent lock requests but inherits MCS performance for frequent requests.

4. Conclusion

Through this we have seen the impact of cache-line bouncing on the various spin lock implementations. And, the usage of thread local spinning in MCS locks provides scalable performance compared to spin lock and ticket lock implementations.

5. References

[1] Mellor-Crummey, John M. and Scott, Michael L. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. In ACM Transactions in Computer Systems, 1991

[2] Wang, Tianzheng and Chabbi, Milind and Kimura, Hideaki. *Be My Guest: MCS Lock Now Welcomes Guests.* In PPoPP '16.

[3] https://lwn.net/Articles/590243

[4] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf