README

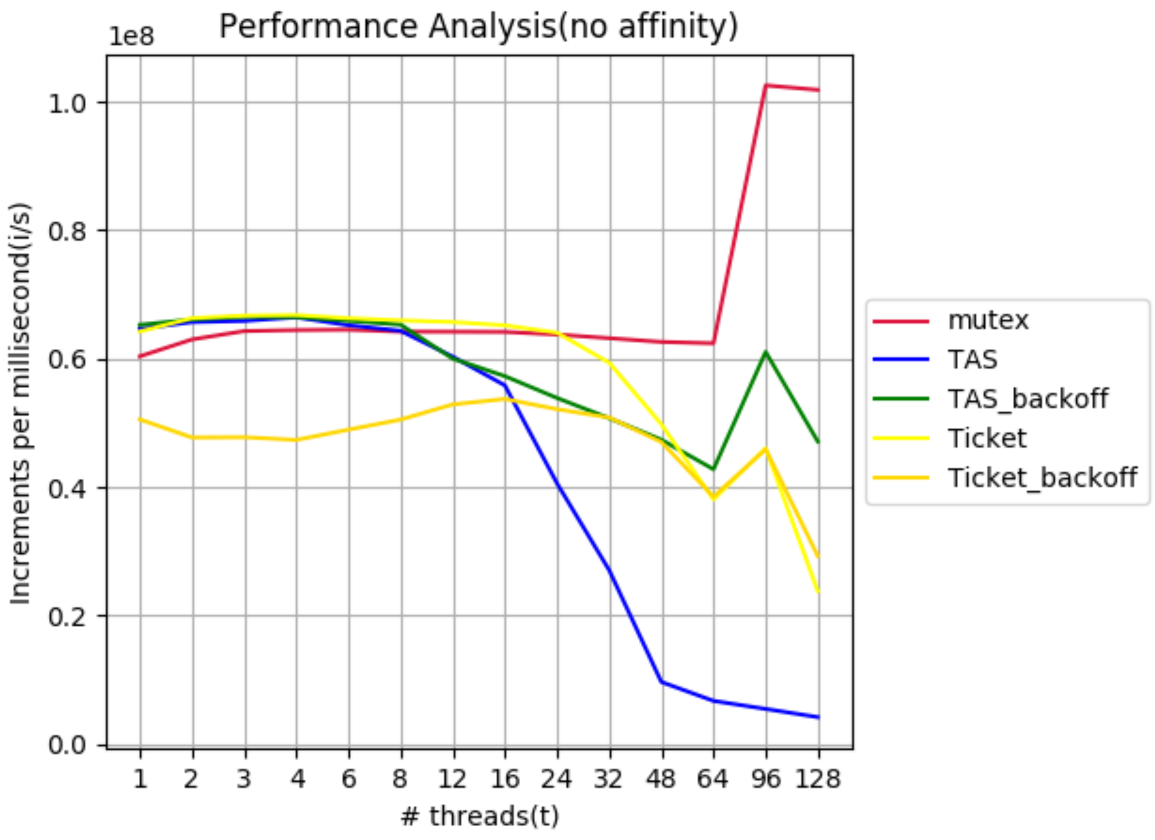
Zhenfei Ji
zji@ur.rochester.edu

Format of input parameters: ./a.out "-t 5" "-i 50000"

In assignment 1, we are required to implement nine different kinds of locks:

(1) C++ mutex lock
(2) Test_And_Set lock
(3) Test_And_Set lock with exponential backoff
(4) Ticket lock
(5) Ticket lock with proportional backoff
(6) MCS lock
(7) K42 MCS lock
(8) CLH lock
(9) K42 CLH lock

With increments 60000 and the number of threads pick from 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, and 128, the performance graph that shows increments per millisecond as a function of thread count for each lock is as following (For sake of convenient observation, I separate these nine locks into two graphs):
The result is collected on IBM Power machine.



(1) A Mutex, in its most fundamental form, is just an integer in memory. This integer can have a few different values depending on the state of the mutex. There are two fundamental operations which a mutex provides: lock and unlock.
A thread wishing to use the mutex, must first call lock, then eventually call unlock to release it. There can be only one lock on a mutex at any given time. The thread holding the lock is the current owner of the mutex. If another thread wishes to gain control, it must wait for the first thread to unlock it.
From the above graph, we can see that with the number of threads increasing, the time consumption per is also increasing. We can explain this result from perspective of the memory. If two threads try to lock the same mutex at the same time, the processors that are running those threads have to communicate and deal with ownership of the relevant cache line, which greatly slows down the mutex acquisition. Also, one of the two threads will have to wait for the other thread to execute the code in the critical section and then release the mutex, further slowing down the mutex acquisition for one of the threads.

(2) The Test-And-Set (TAS) Lock is the simplest possible spinlock implementation. It uses a single shared memory location for synchronization, which indicates if the lock is taken or not. The memory location is updated using a test-and-set (TAS) operation. TAS atomically writes to the memory location and returns its old value in a single indivisible step. TAS lock is not fail as it doesn't guarantee FIFO ordering among the threads competing for the lock.
While the TAS is very easy to implement, its scalability is very bad. Already with just a few threads competing for the lock, the amount of the required cache line invalidations to acquire or release the lock quickly degrades performance. There are two aspects of this problem:

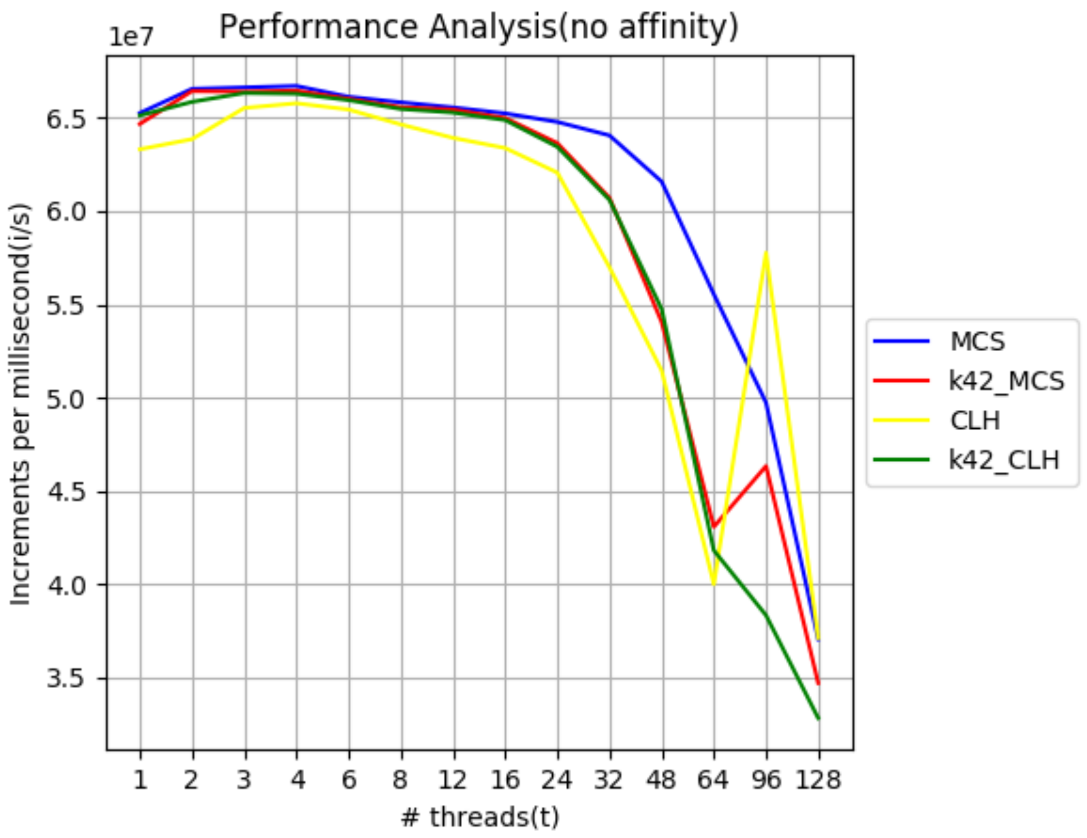
- *std::atomic_bool::exchange()* always invalidates the cache line that *Locked* resides in, regardless of whether it succeeded in updating *Locked* or not.
- When the spinlock is released, all waiting threads simultaneously try to acquire it, which means first invalidating the cache line copy of all threads waiting for the lock and then reading the valid cache line copy from the core which released the lock. With *t* threads this results in O(t) memory bus transactions.

Though the result from graph shows that there is a negative relation between the number of threads and time consumption, it should be the penalty of the contention on the hardware.

(3) With the number of threads grows large, there would be more threads seeing the lock is free but fail acquiring it subsequently because some other thread was faster, that is high contention for the lock. In this situation, a TAS lock with exponential backoff means waiting for a short time to let other threads finish before trying to enter the critical section again. Waiting a short time without trying to acquire the lock reduces the number threads simultaneously competing for the lock. The larger the number of unsuccessful retries, the higher the lock contention and the longer the thread should back-off.
The downside of waiting is that it renders the lock even unfairer than it is already. Threads that have been attempting to acquire the lock longest are also backing-off longest. Consequently, newly arriving threads have a higher chance to acquire the lock than older threads. In addition, the result also shows that the overall time consumption is bigger than the TAS lock without back-off.

(4) The Ticket lock is similar to TAS lock, but it supports FIFO fairness. The speed and scalability of ticket lock and spinlock is almost the same comparing to scalable lock like MCS lock. Both of them introduce lots of cache line invalidation and memory read which overwhelms the memory bus.
Under the condition of no affinity, the time consumption goes up with the increment of the number of threads. Besides, the speed of ticket lock is slower than TAS lock since in an unfair lock, the thread that hogs the lock early finishes more quickly, giving the scheduler less work to do.
The biggest disadvantage of the ticket lock is that the fairness property backfires once there are more threads computing for the lock than there are CPU cores in the system. The problem is that in that case the thread which can enter the critical section next might be sleeping. This means that all other threads must wait, because of the strict fairness guarantee.

(5) Under high content the ticket lock backs-0ff proportionally. The reduced amount of memory bus traffic improves the performance of ticket lock. In graph we can see that the speed of ticket lock with proportional back-off is indeed faster than naïve ticket lock.



(6) The previous locks like ticket lock and TAS lock, the traffic generated by their acquisition is linear to the number of threads competing for the lock. MCS lock improves this by applying a queue which contains a node for each CPU waiting on the lock. Every CPU which wants to wait on the lock allocates a queue node, containing a link (to the next in queue) and a Boolean flag. FIFO ordering is guaranteed by the

queue, and each CPU can spin in a variable which is thread-local. Therefore, MCS lock variants only trigger *O (1)* many cache line invalidation.

From the graph, we can see that the performance of MCS lock depends on the number of threads: the more the threads engage in contention, the faster it is. There are 160 CPUs and 20 cores in IBM Power machine, and because of no affinity, the scheduler manages to place each thread onto a different CPU and keep it there. We also know that MCS invariants only trigger *O (1)* many cache line invalidation, and hence, compared with TAS-based spinlocks which invalidate *O(#threads)*, MCS is indeed faster especially when the number of waiting threads is large.

(7) One disadvantage of the MCS lock is the need to pass a qnode pointer to acquire and release. The K42 version of MCS lock omits the extra parameters and can be substituted in without rewriting all the call points. The performance of K42 MCS lock is basically similar to MCS lock except that the space needs of it is one more than MCS lock.

(8) The CLH lock performs a little slower than MCS lock from the graph, which can be explained by a lot of remote memory references, i.e., spinning is performed on a remote variable.

(9) The K42 version of CLH lock performs similar to CLH lock. The only real difference is that instead of requiring the caller to pass a qnode to release, we leave that pointer in a head field of the lock. No dynamic allocation is required: the total number of extent nodes is always n + j for n threads and j locks.