locks

Naive test and set lock

The Naive test and set lock is the samplest implementation of spin lock. It use a `atomic_flag` to do `test_and_set` function to implement the acquire process.

```
void acquire(){
        while (flag.test_and_set());
        atomic_thread_fence(std::memory_order_acquire);
        atomic_signal_fence(std::memory_order_acquire);
    }
    void release(){
        flag.clear();
    }
```

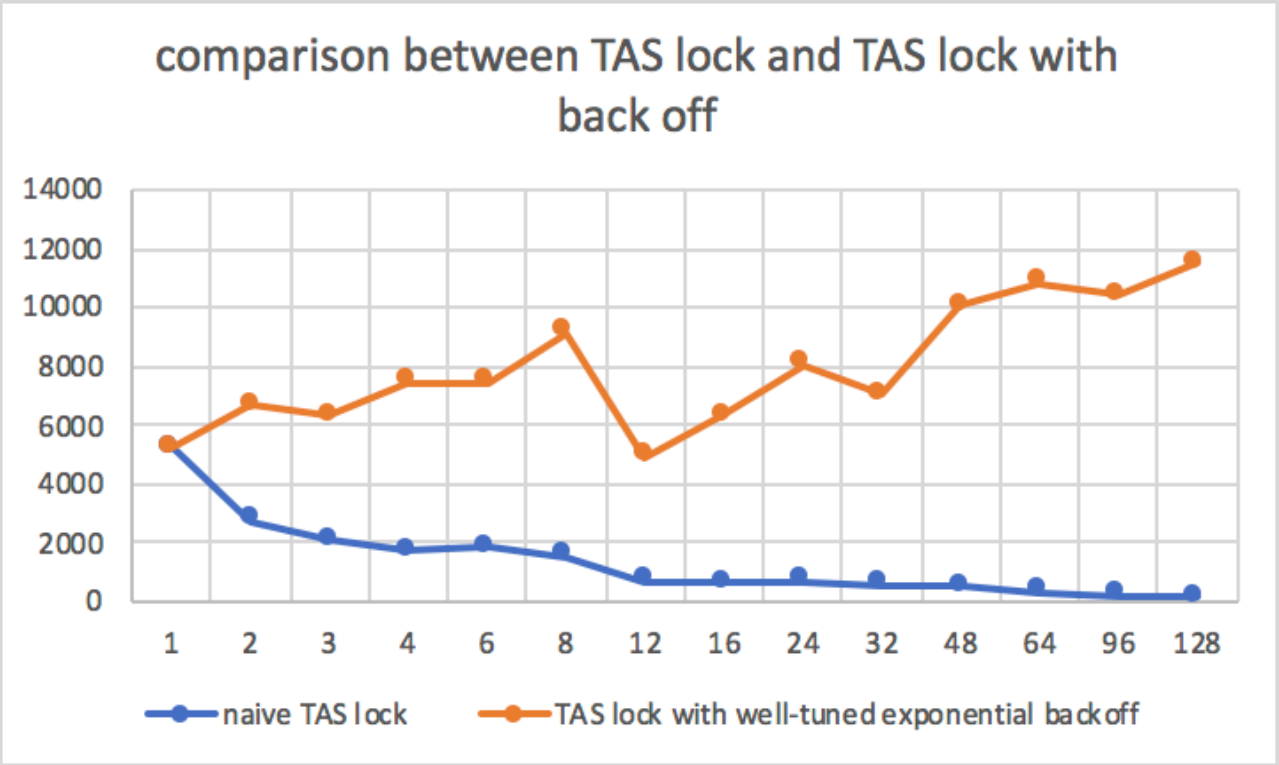
`flag.test_and_set()` will set the value of flag to *true* and return the original value of flag. If the return value is true (which means that some other thread has already set the flag) it should repeat this `test_and set()` operation until some thread release this lock through `flag.clear()`.

TAS lock with well-tuned exponential backoff

The main problem of test and set lock is that every threads are keeping trying do `test_and_set` operation, which will increase the data race between them. The test and set lock with well-tuned exponential backoff let the threads who are not `test_and_set()` successfully do something else instead of keep aquiring the lock.

```
void acquire(){
        int deley = base;
        while(flag.test_and_set()){
            auto time_first = std::chrono::high_resolution_clock::now();
            while ((std::chrono::high_resolution_clock::now() -
time_first).count() < deley);
            deley = min(deley*multiplier,limit);
        }
        atomic_thread_fence(std::memory_order_acquire);
        atomic_signal_fence(std::memory_order_acquire);
    }
```

There are three thing which decide how will a thread wait, `base` , `multiplier` and `limit` . In my idea it is a try process. Firstly, the thread will try to wait for a short time(base), if it still can not acquire the lock, then it will wait for more time(deley*multiplier). But the waiting time can not always be increased, There is a `limit` to limit the waiting time. I use a while loop instead of `sleep` because the backoff time is too short and it cost to much to weak up thread which is sleeping. I try various of `base` , `multiplier` and `limit` meking the delay time varies from 10 to 1000 nanoseconds, Eventually identified three proper value for those three parameters(10000,1000000,2)



From the flow chat above, we can find the performance of TAS lock with back off improve much compared with TAS lock without backoff.

naive ticket lock

The test and set lock can avoid the deadlock but can not ensure the fairness. There may be some thread can alway get the lock but some can not. The ticket lock give the threads which attempt to aquire a number. The lock can record the which thread is holding the lock and who is the next.

```
void acquire(){
        int my_ticket = next_ticket.fetch_add(1);
        while(my_ticket != now_serving.load());
        atomic_thread_fence(std::memory_order_acquire);
        atomic_signal_fence(std::memory_order_acquire);
    }
void release(){
        now_serving.fetch_add(1);
    }
```

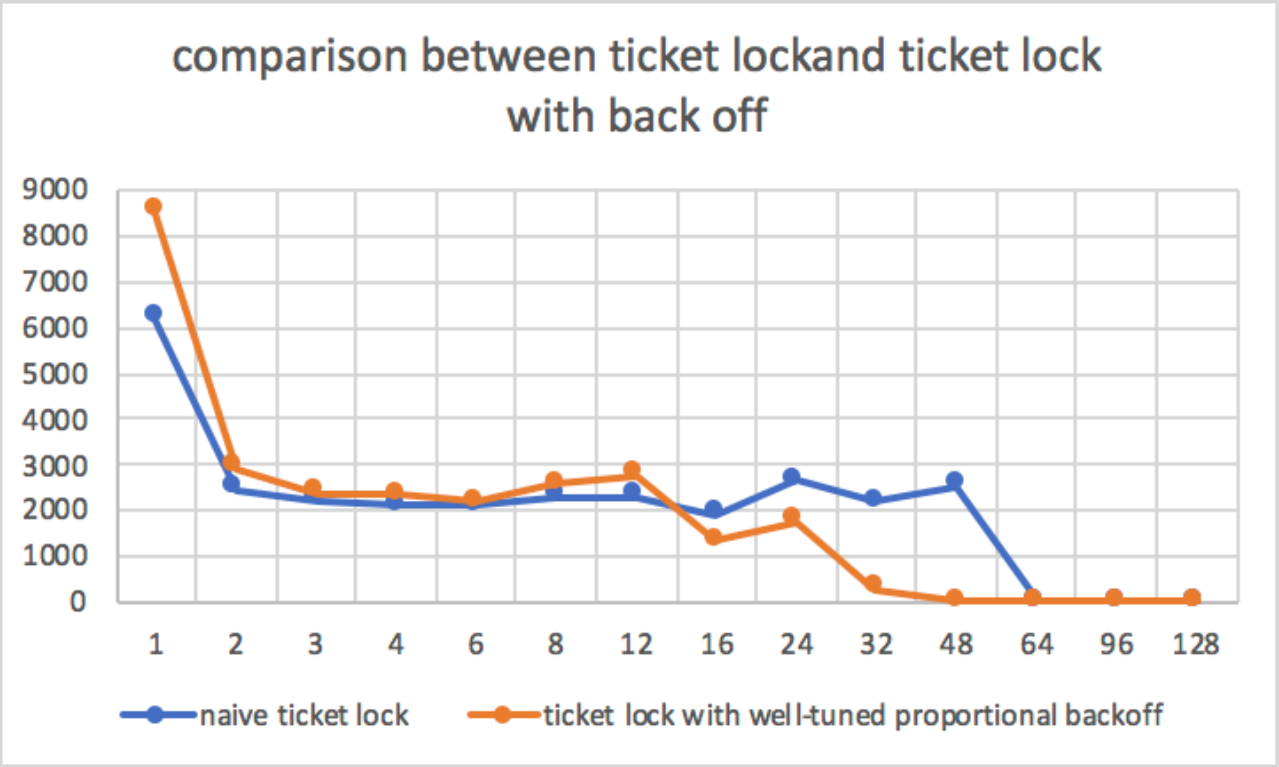
The variables `next_ticket` and `now_serving` are all atomic variable. That is because I want to use `fetch_add` operation to make sure that `next_ticket` and `now_serving` can not be add more than 1 at a time.

ticket lock with well-tuned proportional backoff

The naive ticket lock has the same problem with naive test and set lock. Each thread will always keep attempting acquire the lock, which will result in more data race. Similarly i set a backoff time for the naive ticket lock.

```
 void acquire(){
        int my_ticket = next_ticket.fetch_add(1);
        while(my_ticket != now_serving.load()){
            auto time_first = std::chrono::high_resolution_clock::now();
            while ((std::chrono::high_resolution_clock::now() -
time_first).count() < base);
        }
```

There is a parameter to be tuned, `base`. I try the nunber between 1 and 1000. I find that when the base is lower than 100, the result will fluctuates. When the value of base is larger than 100, the result will became worse as thebase increase. So I choose 25 as the parameter value.



From the flow chat above, we can find in this time the performance improve just a little. In my opinion, this is because the cost of `load()` operation is fewer.

MCS lock

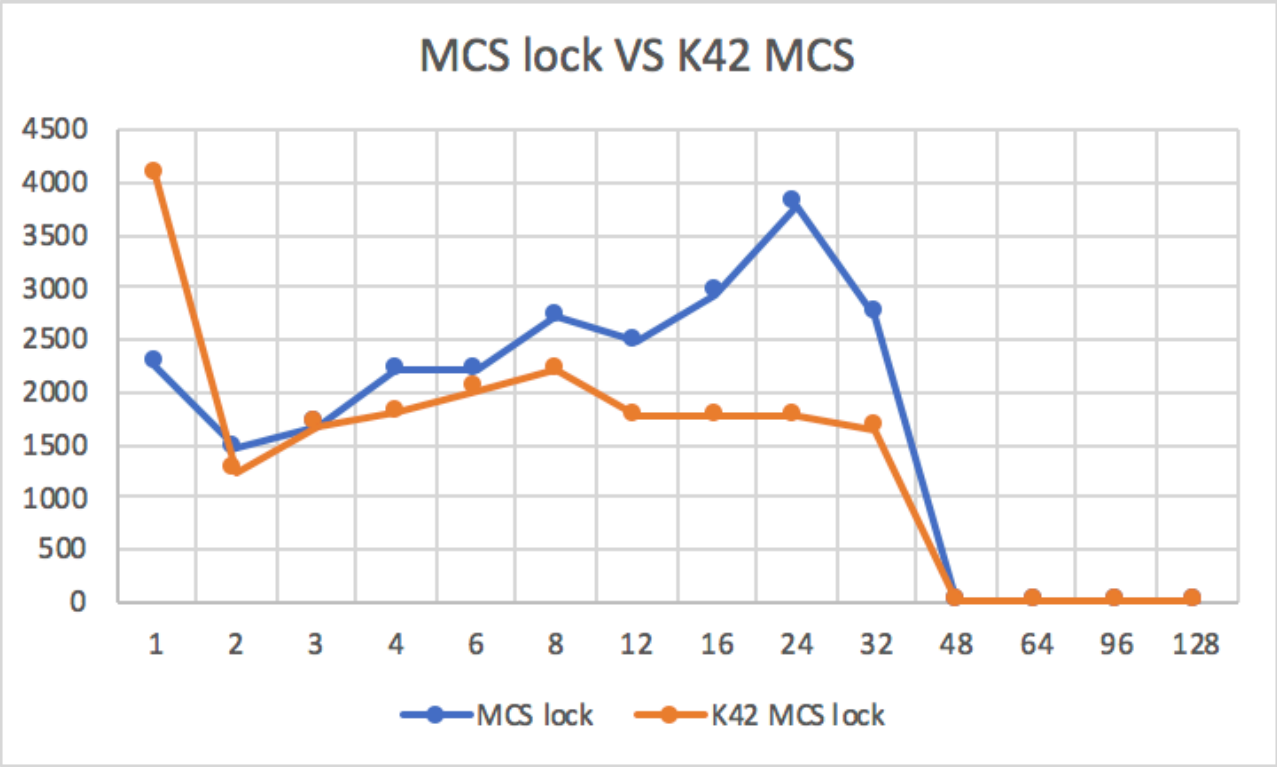
In the first four locks, each threads will frequently read and write the same variable(flag), each read and write operation must be cache synchronized between multiple processor caches, which will result in heavy system bus and memory traffic. Reduce overall system performance. The MCS lock is implemented based on queue. All the threads attempting to acquire the lock are in a queue. When a node's predecessor is null, it can get the lock successfully. To specific, when a thread want to acquire the lock, it should load the `tail` of the lock as its predecessor. Then the thread add a new node to the queue after the tail, and use `exchange` operation to exchange the value between `tail` and the new node. When a thread want to release the lock, if the next node of self is `null`, which means no one wants to get this lock, it will just set the node of self to `null`. if the the next node of self is not `null` , which means some one wants to get this lock, the thread will set the variable `waiting` to **true** value, which note other threads that they can acquire this lock.

```
void acquire(Qnode &q){
        q.next.store(nullptr);
        q.waiting.store(true,std::memory_order_relaxed);
        Qnode* prev = tail.exchange(&q,std::memory_order_release);
        if (prev != nullptr){
            prev->next.store(&q,std::memory_order_relaxed);
            while (q.waiting.load()){
            }
```

```
        }
        atomic_thread_fence(std::memory_order_acquire);
        atomic_signal_fence(std::memory_order_acquire);
    }
    void release(Qnode &q){
        Qnode* succ = (q.next).load(std::memory_order_acquire);
        if (succ == nullptr){
            Qnode *tmp = &q;
            Qnode *_null = nullptr;
            if (tail.compare_exchange_strong(tmp, _null)){
                return;
            }
            while(succ == nullptr){
                succ = q.next.load(std::memory_order_relaxed);
            }
        }
        succ->waiting.store(false);
    }
```

K42 MCS lock

A weak point fo MCS lock is it need pass a node as a parameter to `aquire` and `release` . So The K42 MCS lock move the queue into the lock itself. The idea is the same as MCS lock. The picture below show the throughput(measured in increments per millisecond) of *K42 MCS lock* and *K42 MCS lock* as the number of threads increase.

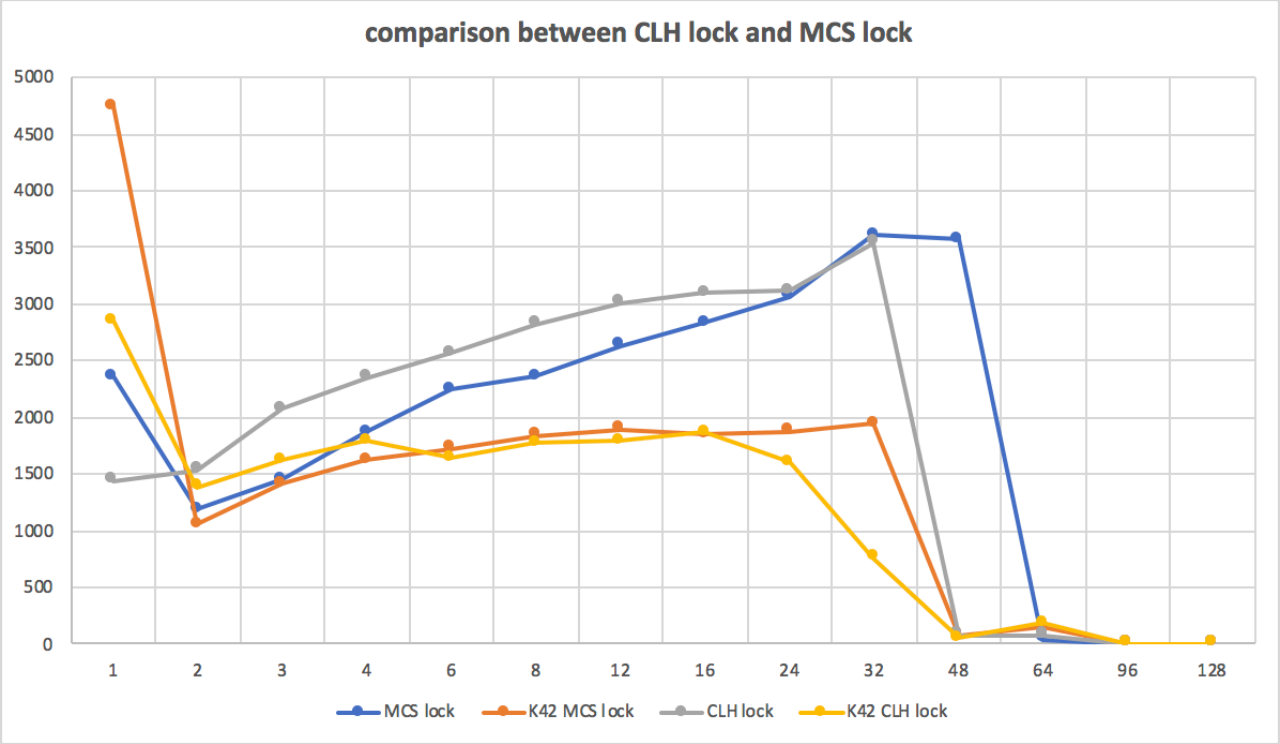


From the chat above, we can find that the performance of the two locks have the same trend. But the k42 MCS lock is a little slower than MCS lock. That is because the K42 MCS lock will have much more operation to maintain the whole queue in the lock class.

CLH lock

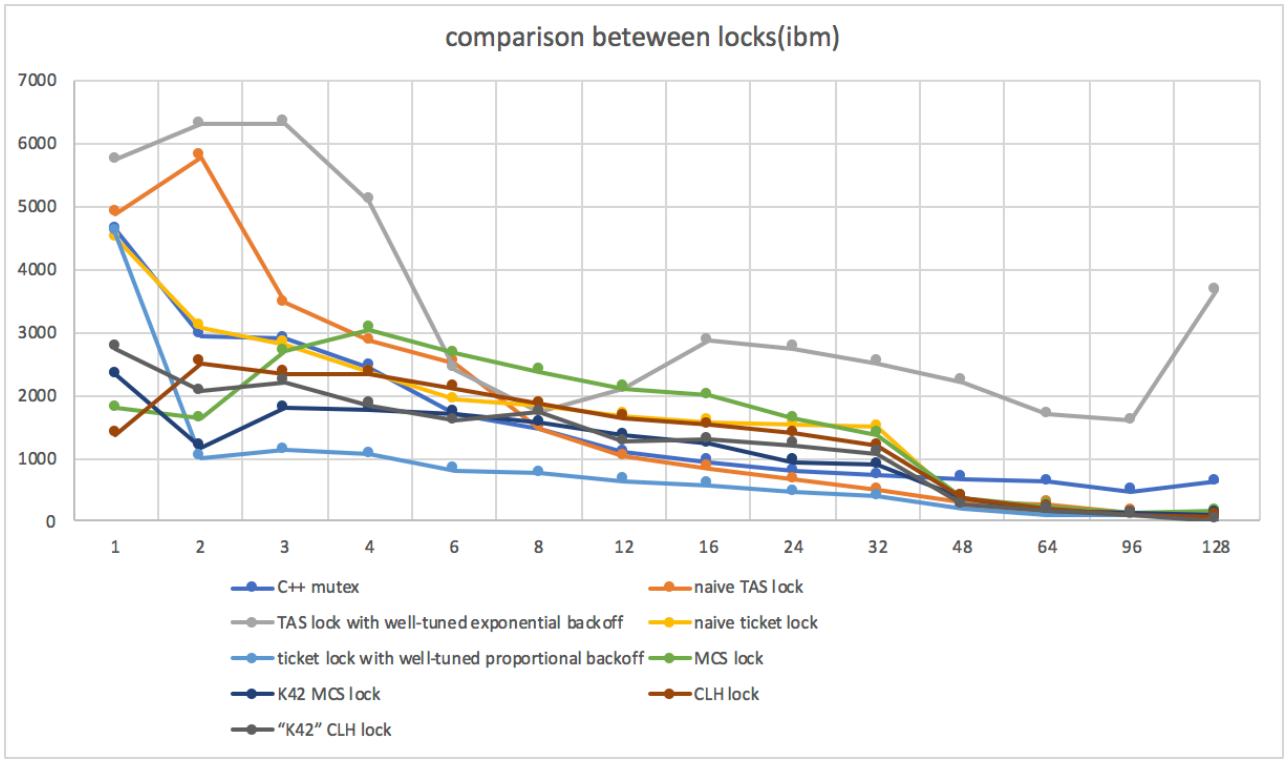
The CLH lock is also a queue based lock. But there is some different between CLH lock and MCS lock. The CLH lock is spinning on the previous node, and waiting to acquire the lock. In this case, different threads has their own cache. if There are some distance in cache between the previous node and self, frequently check whether the previous node has released the lock cost too much.

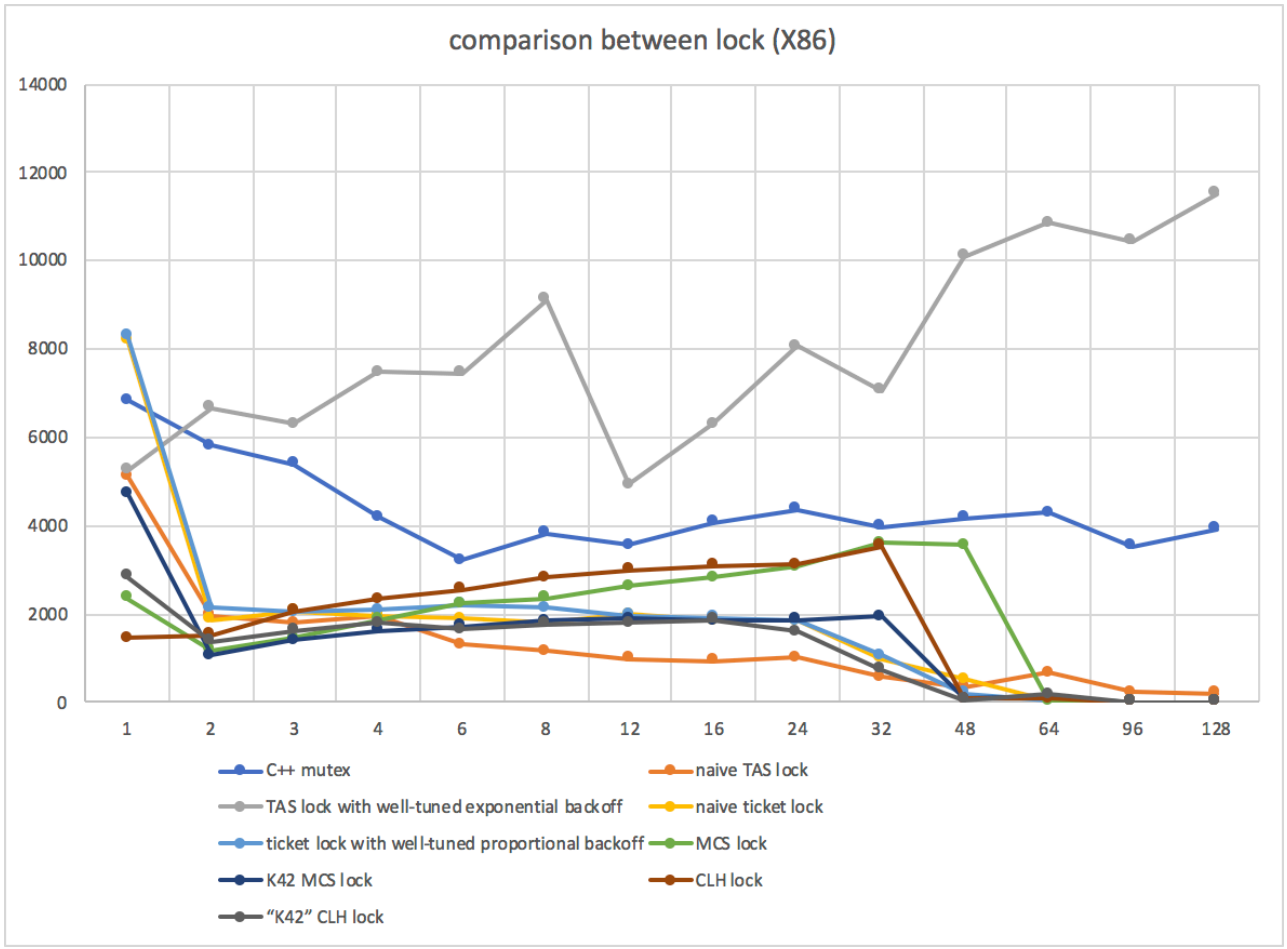
The CLH lock have the same problem like MCS lock. That is the `acquire()` and `release()` function need parameters.The K42 CLH lock Make up for this shortcoming by moving the waiting queue into the lock.



From the flow chat above, we can find there is a small problem that CLH almost have greater performance than MCS lock.

Experiment on IBM machine


comparison beteween locks(ibm)
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
6
8
12
16
24
32
48
64
96
128
C++ mutex
naive TAS lock
TAS lock with well-tuned exponential backoff
naive ticket lock
ticket lock with well-tuned proportional backoff
MCS lock
K42 MCS lock
CLH lock
"K42" CLH lock


comparison between lock (X86)
14000
12000
10000
8000
6000
4000
2000
0
1
2
3
4
6
8
12
16
24
32
48
64
96
128
C++ mutex
naive TAS lock
TAS lock with well-tuned exponential backoff
naive ticket lock
ticket lock with well-tuned proportional backoff
MCS lock
K42 MCS lock
CLH lock
"K42" CLH lock

From the experiment on two different machine, we can conclude that in both machine, the TAS lock perform the best. In the power machine almost every lock will became slower when threads grow, except TAS lock with backoff. Due to that the ibm machine have 160 hardware threads, all lock perform better on it. That is because the TAS lock doesn't have too much extra operation, and also doesn't need complex data struct to record the waiting queue. But the disadvantage of TAS lock is that it can not guarantee the fairness. In x86 machine, With the threads increase the speed of MCS lock and CLH lock is become faster than TAS lock and ticket lock. I think that is because In TAS lock and ticket lock more threads will result in more frequent accessing to the same memory. However, in MCS lock and CLH lock every threads just care its predecessor.