Contents

Project 4: Locks and Threads

JuranDing

2018-12-07

1 Wrapper

Before I realize a lock, there are some preparations for the lock implementation like atomic operation and futex wrapper.

1.1 Atomic Operation

The atomic operation is not implemented with pure C language but with assemble language and some inner built system call by the support of x86 architecture. And I list some atomic operations that will be used in the lock-make program.

- Assemble Language Operation

  - **xchg**
    In the following code, *asm volatile* means that the sentence in the brackets should be explained as the assemble language. The sentence is generally divided into 4 sectors in the extended Asm. The first is *instruction*, and the second is *output* operation while the third is *input*. The first 3 sectors are easy to understood. The 4th is called *Clobber List* which is used to inform gcc that we will use and modify some hardware registers ourselves and the value loaded into these registers won't be valid. Like the *cc*, it tells gcc that instruction can alter the condition code register to avoid some unpredictable conditions. And the specific explanation is in the code's annotations.

```
static inline uint
xchg(volatile unsigned int *addr, unsigned int newval)
{
        uint result;
        //lock means that the operation is atomic
        //xchgl means exchange values
        asm volatile("lock; xchgl %0, %1" :
        /*
         * "+m" "+" is a modifier means that operation
```

```
          * for the varaible is reading or writing
          * "m" is memory constrain, the variable should
          * be a memory
          */

         /*
          * "=a", "=" means that the variable is write-only
          * "a" is a kind of register starting with "a"
          */
                 "+m" (*addr), "=a" (result) :
         /*
          * "1" means that the valuable mathching the number
          * in the instruction
          */
                 "1" (newval) :
         /*
          *  As the condition codes will be changed,
          *  we are adding "cc" to clobberlist.
         */
                 "cc");
         return result;
}
```

- Inner-Built Function

  The following builtins are intended to be compatible with those described in the Intel Itanium Processor-specific Application Binary Interface. In most cases, these builtins are considered a full barrier. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

  - **__sync_val_compare_and_swap(type *ptr, type oldval type newval)**

    That is, if the current value of *ptr is oldval, then write newval into *ptr. There are some tricks to use the operation which is about return value. When the comparison is successful the value returns one while it returns the content *addr*. So if the original value of addr is 1, no matter the comparison successful or failed, it will return true. The characteristic can be used to reduce code efficiently.

```
\_\_sync\_val\_compare\_and\_swap(type *ptr,
         type oldval type newval)
{
         if(*ptr == oldval)
         {
                 *ptr = newval;
                 return 1;
         }
         else return *addr;
}
```

– **__sync_lock_test_and_set (type *ptr, type value, ...)**

  This builtin, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes value into *ptr, and returns the previous contents of *ptr. The deference between the *test and set* and the *swap and compare* is that the latter doesn't need a compare, just set the value to the *addr*. Here's the programming logic with pseudocode:

```
__sync_lock_test_and_set (type *ptr, type value, ...)
{
        tmp  = *ptr;
        *ptr = value;
        return tmp;
}
```

– **__sync_fetch_and_add**

  It's used to implement a atomic ++. It begins fetch before the specific arithmetic operations or the logic operation. Some other arithmetic operations or the logic operations are applied to it too. Here's the programming logic with pseudocode:

```
\_\_sync\_fetch\_and\_add(type *addr, type value)
{
        tmp = *addr ;
        *addr += value;
        return tmp;
}
```

– **__sync_sub_and_fetch**

  It's used to implement a atomic - -. It begins the specific arithmetic operations or the logic operation before fetch. Some other arithmetic operations or the logic operations are applied to it too. Here's the programming logic with pseudocode:

```
\_\_sync\_fetch\_and\_add(type *addr, type value)
{
        tmp = *addr -value ;
        *addr -=  value;
        return tmp;
}
```

1.2 Futex Wrapper

To implement the mutex, we need to stop the thread for a while (let it sleep), and the linux system provides us the function to sleep the thread called futex. The parameters passing is trivial so we need a wrapper.

1.2.1 Futex

Just paste the code:

```
int futex(int *uaddr, int futex_op, int val,
        const struct timespec *timeout, int *uaddr2, int val3)
{
        return syscall(SYS_futex, uaddr,
                futex_op, val,timeout, uaddr, val3);
}
```

1.2.2 Futex_Wait

This operation tests that the value at the futex word pointed to by the address uaddr still contains the expected value val, and if so, then sleeps waiting for a FUTEX_WAKE operation on the futex word. The purpose of the comparison with the expected value is to prevent lost wake-ups.

```
void futex_wait(int * mtx,int v)
{
        futex(mtx,v,2, NULL, NULL, 0);
}
```

1.2.3 Futex_Wake

This operation wakes at most *val* of the waiters that are waiting on the futex word at the address *uaddr*.

```
void futex_post(int *mtx)
{
        futex(mtx, FUTEX_WAKE, 1, NULL, NULL, 0);
}
```

2 Spin Lock

2.1 Implementation

Spin Lock's implementation is simple. Here I give several versions using different strategy in busy-waiting the performance comparison.

- **Version0: Just Query**
  Just keep asking the current state of the lock, if it's unlocked, lock it.

```
void spinlock_acquire(spinlock_t* inLock)
{
        while(xchg(inLock,1));
}
```

- **Version1: pause for a while**
  In version0, there's a serious problem. The cpu which is blocked continuously asks memory for the current state of the lock. It wastes too much resource of the bus bandwidth and each waiting thread of the same priority is likely to waste its quantum spinning until the thread that holds the lock is finally finished. Waiting for a relatively longer time can slightly alleviate the first problem. So just wait a while.

```
#define SYSPAUSE    asm("pause":::"memory")
void spinlock_acquire(spinlock_t* inLock)
{
        while(xchg(inLock,1))SYSPAUSE;
}
```

- **Version2: roll query**
  Version1 solved the 1st problem, but it's weak on the 2nd problem too. So we need a more delicate solution: when we ask for the critical zone and fail, we let the thread relinquishes the cpu, and at the same time other threads can takes up the cpu quickly it perfectly solves the second problem. And all we need to do is call a system function : *Sched_yield()* causes the calling thread to relinquish the CPU and the thread is moved to the end of the queue for its static priority and a new thread to run.

```
void spinlock_acquire(spinlock_t* inLock)
{
        while(xchg(inLock,1))sched_yield();
}
```

- **Version3: A Fairer Way**
  In order to fairly distribute system resource, it seems like a good idea that every one goes into the critical zone one by one. *Ticket* is a good method on the basis of the idea. It's similar to bank service. Every customer comes to the bank gets his number first and then wait the window yells out his number.

```
void spinlock_acquire(spinlock_t* inLock)
{
        unsigned iam = ATMXADD(&(inLock->user),1);
        while(inLock->ticket!=iam);
}
void spinlock_release(spinlock_t* inLock)
```

```
{
        inLock->ticket++;
}
```

2.2 Performance Comparison

- counter function comparison
  Here's the test code: *counter increments 5,000,000 and decrements 5,000,000*

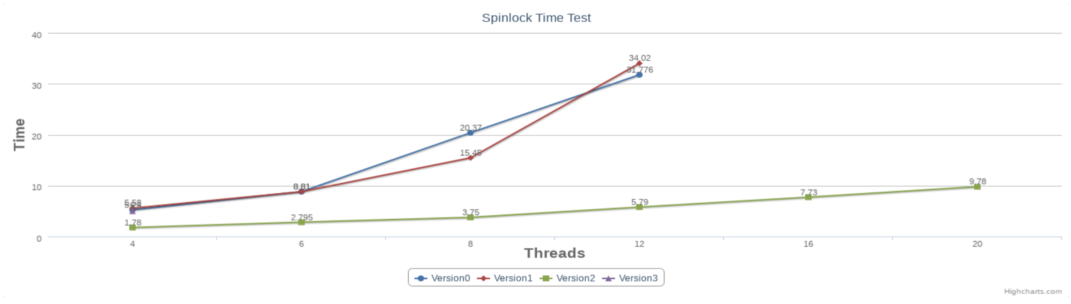
```
#define COMPLEX 5000000
printf("Thread %d,%d\n",id,counter_get_value(ct));
for(int i=0;i<COMPLEX;i++)
        counter_increment(ct);
printf("Thread %d,%d\n",id,counter_get_value(ct));
for(int i=0;i<COMPLEX;i++)
        counter_decrement(ct);
printf("Thread %d,%d\n",id,counter_get_value(ct))
```

And here's the result:

Threads	Version0	Version1	Version2	Version3
4	5.24(s)	5.58	1.78	5.00
6	8.81	8.81	2.795	too long
8	20.37	15.45	3.75	too long
12	31.776	34.02	5.79	too long
16	too long	too long	7.73	too long
20	too long	too long	9.78	too long

Table 1: My caption

And here's the line graph:



It's basically accorded with our assumptions. But the TICKET spinlock performs much worse than our expectations. Why? Trying to be fair is good but fairness isn't always coming with efficiency. It does perform better than the *version0* and *version1* because the cpu'c cores' number fits with the thread number. But when there exists a cpu resource race, it performs really worse then *version0* and *version1*. Let's focus on the threads racing in a cpu core. Assuming that 5 threads are competing for cpu core. Thread1 asks for a number 1, Thread2 asks for a number 2..etc. Assuming that now it's thread1' turn, but it's not in the core,(Maybe the costumer 1 goes to the toilet...) the cpu can go on the process when it goes back, it really takes a long time. We improve the fairness to improve the efficiency. But they're not in a linear relationship.

2.3 Add Ons

I'm not satisfied with the `version3`'s performance, so I retry again in a more sophisticated way called `queue_control_lock`. It comes with more memory consumption, but performs as well as the best spinlock.

2.3.1 A concurrent queue

It's a really tricky queue implementation. To finish it, we should define some queue conditions.

- **empty** : When the queue's head meet with queue's tail, empty.
- **dequeue** : Who gets permission is the one who changes the head to head-¿next. We need an atomic operation to safely change the head to the next. And we the element leaves, it should be set to zero to signify that it has gone.
- **enqueue** : A new node comes with value set to 1(`in queue mark`) and next set to NULL. Who gets the permission to the tail is the one who changes the tail-¿next to the new node. Then we should change the tail to tail-¿next. But this time, the thread doesn't have to make sure that it has successfully changed the tail because it's the only one who can changes the tail because it's the only on in the code section which changes the tail.

```
void queue_init(lqueue_t *q)
{
        lnode_t *tmp = (lnode_t*)malloc(sizeof(lnode_t));
        tmp->value = 0; tmp->next = NULL;
        q->head = tmp;  q->tail = tmp;
}
void queue_enqueue(lqueue_t *q,int * val)
{
        lnode_t *tmp = (lnode_t*)malloc(sizeof(lnode_t));
        tmp->value = val;
        tmp->next = NULL;
        lnode_t *r;
        do
        {
                r = q->tail;
        }while(!__sync_bool_compare_and_swap(&(r->next),NULL, tmp));
        __sync_bool_compare_and_swap(&(q->tail),r,tmp);
}
int* queue_dequeue(lqueue_t *q)
{
        lnode_t* p;
        do
        {
                p = q->head;
        }while(!__sync_bool_compare_and_swap(&(q->head),p,p->next));
        *(p->next->value) = 0;
        return p->next->value;
}
int queue_empty(lqueue_t *q)
{
        if(q==NULL || q->head==q->tail )
                return 1;
        return 0;
}
```

2.3.2 Lock

A guard is set to ensure everyone comes to the code set is mutual exclusive. But after futex-sleep, the code section is not guarded. So we need a new flag to ensure it, so we reuse the flag to guard the critical zone. After performance test, the efficiency is close to the fastest spinlock.

```
void spinlock_init(spinlock_t* inLock)
{
        inLock->flag = 0;
        inLock->guard = 0;
        inLock->q = (lqueue_t *)malloc(sizeof(lqueue_t));
        queue_init(inLock->q);
}
void spinlock_acquire(spinlock_t* inLock)
{
        while(xchg(&inLock->guard,1) == 1);
        if(__sync_bool_compare_and_swap(&inLock->flag,0,1))
        {
                inLock->guard = 0;
        }
        else
        {
                int *tmp = (int*)malloc(sizeof(int));
                *tmp = 1;
                queue_enqueue(inLock->q,tmp);
                inLock->guard = 0;
                while(!__sync_bool_compare_and_swap(
                                        &inLock->flag,0,1))
                {
                        futex_wait(tmp, 1);
                }
        }
}
void spinlock_release(spinlock_t* inLock)
{
        while(xchg(&inLock->guard,1) == 1);
        if(!queue_empty(inLock->q))
        {
                int *tmp = queue_dequeue(inLock->q);
        futex_post(tmp);
        }
        xchg(&inLock->flag,0);
        inLock->guard = 0;
}
```

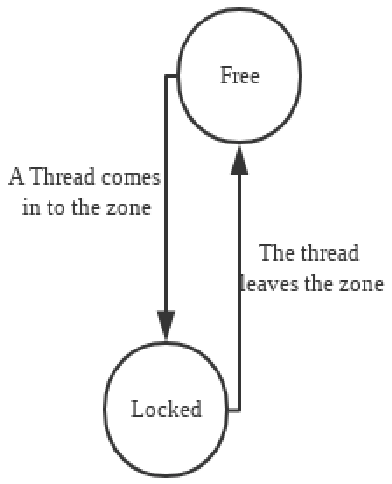
3 mutex Lock

When we need a public resource for a relatively long time, it's too wasted for a thread to keep asking for the critical zone continuously. So we can let it sleep until the other thread releases it. To accomplish it, we need a system function called futex which can be used to awake and sleep the thread.

3.1 Implementation

3.1.1 version1.0

It's a version that is completely derived from the mutex idea without any optimization. When a thread comes to visit the mutex lock, if it's not locked, the thread just locks it. Else, futex_wait and waits for others to awake it. Here's the lock's Finite State Machine.



And here's the code:

```
#define UNLOCKED 0
#define LOCKED   1
void mutex_acuire(mutex_t* mutex)
{
        while(__sync_val_test_and_set(mutex,LOCKED))
                futex_wait(mutex,COMPETE);
}
void mutex_release(mutex_t *mutex)
{
        *mutex = UNLOCKED;
}
```

3.1.2 version1.1

There're only two phases in the *version1.0*, so it's a natural idea to add a new phase before sleeping to improve performance by avoiding some occasions like in 2 threads' race, if both of them get in and get out quickly, it's a heavy burden for system to frequently be trapped into the sleep state and awake the sleep threads again and again. And in my implementation, I divided 3 phases called

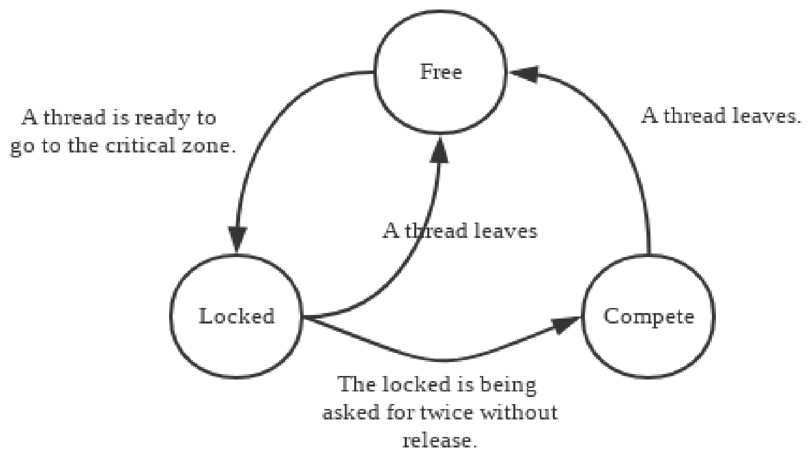
UNLOCKED, *LOCKED*, *COMPETE*.

- *UNLOCKED*
  No one is in the critical zone or a thread just leaves it but no one has asked for the lock.

- *LOCKED*
  A new thread gets into the thread and the lock is free, then before it gets into the critical zone, it lifts the lock to 1 first.

- *COMPETE*
  Now the lock is visited twice without being set UNLOCKED, it tells every thread it's in the *COMPETE* state. Now if someone is asking for the lock, it's just trapped into sleep mode.

Here's the finite state machine:


Free
A thread is ready to go to the critical zone.
A thread leaves.
A thread leaves
Locked
Compete
The locked is being asked for twice without release.

And here's the code:

```
void mutex_acquire (mutex_t* mutex)
{
        //check if mutex is zero, if it is, unlocked to locked
        //state==2 means that it's congesting now state == 1
        //means that it will be locked in next visit
        int state = __sync_val_compare_and_swap(mutex,
                UNLOCKED,LOCKED);
        //if not locked
        while(state)
        {
                //if it is now congested or locked
                if(state == COMPETE || __sync_val_compare_and_swap(
                                            mutex, LOCKED, COMPETE))
                        futex_wait(mutex,COMPETE);
                state = __sync_lock_test_and_set(mutex, LOCKED);
}
```

3.1.3 version1.2

Now there's a choice that we should pay our attention on is that a thread just being awakened should goto which kind of state, *UNLOCKED* or *COMPETE*? Both seem plausible, so we need some experiments.

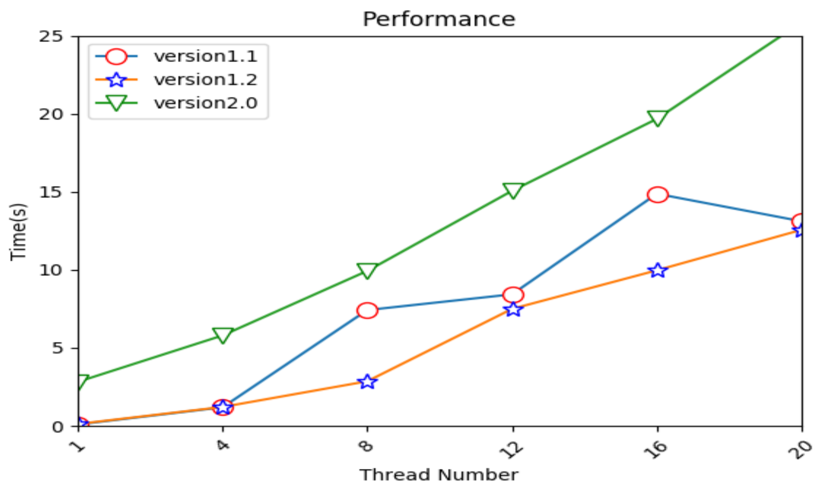
3.1.4 *version2.0*

It's the version implemented in the old *glibc*. The idea of it is that the top bit is used to signify the mutex state(*UNLOCKED*, *LOCKED*) and the value is used to tell its ID. But in my implementation, it performs worse than *version1.0*. It's because that some *futex_post(s)* may be lost to awake a sleeping thread and the bitset operation is an enormous expense.
Here's the code.

```
void mutex_acquire (mutex_t* mutex)
{
        if (atomic_bit_test_set (mutex,31)==0)
                return;
        atomic_increment(mutex);
        int v;
        while(atomic_bit_test_set (mutex,31)!=0)
        {
                v = *mutex;
                if( v >= 0 ) continue;
                futex_wait (mutex,v);
        }
        atomic_decrement (mutex);
        return;
}
```

3.2 Performance

Here's the comparison, from which we can clearly say that the **version1.2** is a winner.



4 Tow phase Lock

Actually, the *version1.1* and *version1.2* is a combination of spinlock and mutex lock. But to adjust the parameter is essential to improve the performance, so we need to complete our code. Here's a simple implementation.

```
void twophase_acquire(twophase_t *mylock)
{
        int state = __sync_val_compare_and_swap(
                                mylock,UNLOCKED,LOCKED);
    while(state)
        {
                if(state == COMPETE || __sync_val_compare_and_swap(
                                        mylock, LOCKED, COMPETE))
                        futex_wait(mylock,COMPETE);
                for(int j=0;j<10;j++)
                {
                        state = __sync_val_compare_and_swap(mylock,
                                        UNLOCKED, COMPETE);
                        if(!state)return;
                }
        }
}
void twophase_release(twophase_t *mylock)
{
        if( __sync_lock_test_and_set(mylock,0) == COMPETE)
                futex_post(mylock);
}
```

The comparison comes in after we finish our data structure.

5 Data Structure

5.1 Counter

My version is simple, just count. (Approximate counter is scalable but not accurate)

5.2 List

The list is simple but where should we set the clock? Each element or the whole list? Assuming that we give every element a lock, then when we lookup the whole list, we need to acquire and release every element in the whole list as well as in the insert and delete operation which is unreasonable. So forget it, just give the whole list a lock.
Here's the data structure but without function which is easy to understand and simple:

```
typedef struct list_value
{
        unsigned int       key;
        struct list_value* next;
}list_value;

typedef struct list_t
{
        lock_t lck;
        list_value* root;
}list_t
```

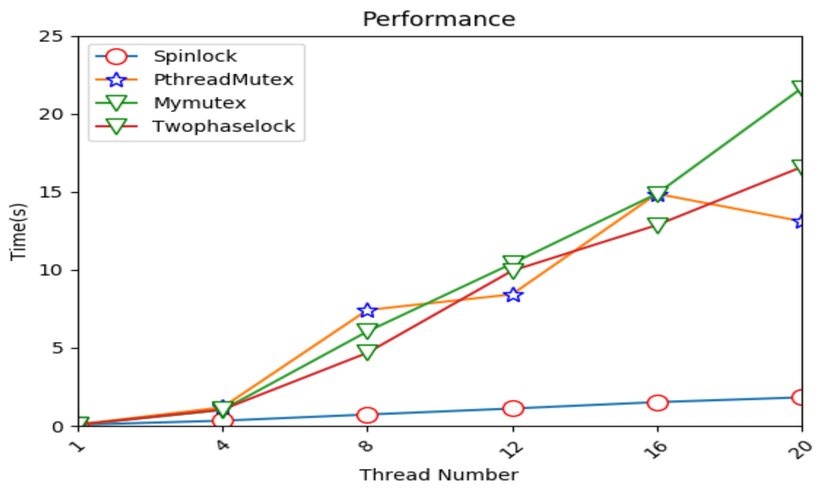
5.3 Hash

It maintained some buckets and every bucket is a *list*, so insert, delete and lookup operations are just calling the list function I implement before in the specific bucket. To be scalable, using a point to maintain memory instead of an array. Here's the data structure and *init* function:

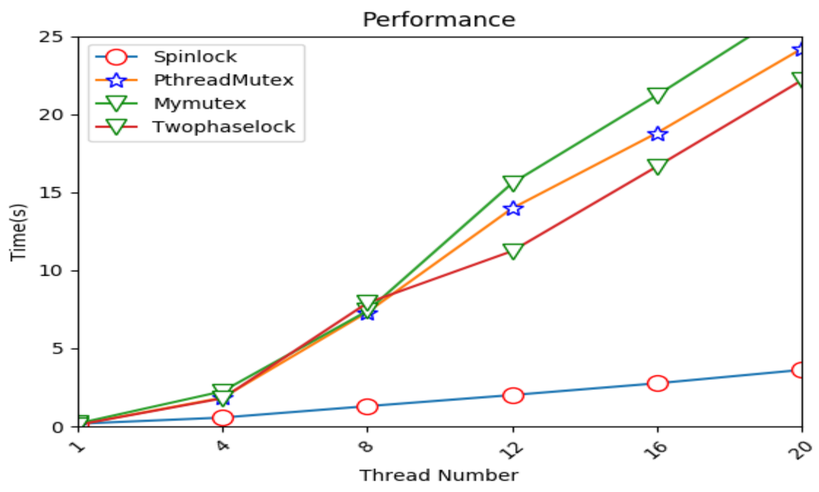
```
typedef struct hash_t
{
        int size;
        list_t *bucket;
}hash_t
void hash_init(hash_t *hash, int size)
{
        hash->bucket = (list_t*)malloc(sizeof(list_t)*size);
        hash->size=size;
        int i=0;
        for(;i<size;i++)
        {
                list_init(&(hash->bucket[i]));
        }
}
```

6 Performance

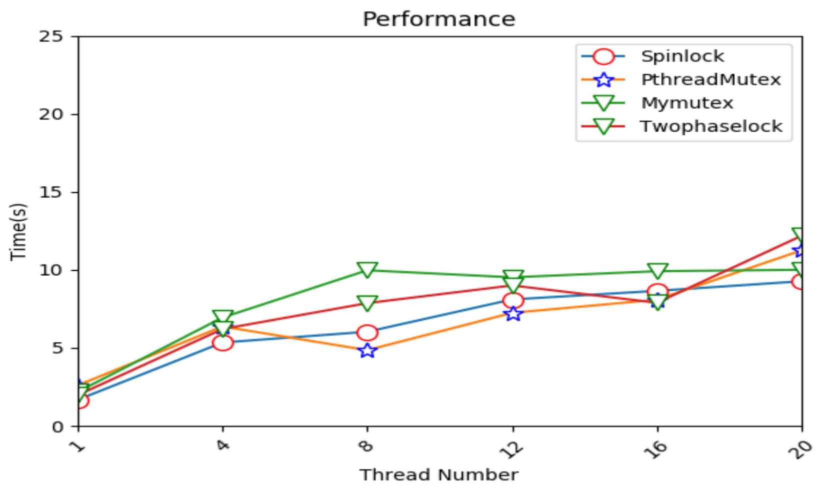
6.1 in counter



6.2 in list



6.3 in hash



From the diagram, we can clearly see that in the counter and list function, spinlock performed far better then mutex whatever how they were implemented. But in hash table, pthread mutex performes better. Spinlock is really powerful, then why actually we mutex is used wider then spinlock? It's because a well-implemented parallel program is visiting implementing critical zone as less possible. So a program like increasing a counter 1e6 doing nothing else in each thread is meaningless, so after several asking, just sleep the thread is much better then keeping asking. It's not only about the cpu's computing resource's waste but also the bus bottleneck. When a cpu keeps asking for bus control, it sometimes blocks other cpu’s visit.