《并行编程作业：读写锁》

一、 问题描述

1. 利用 pthread 其他同步机制（如信号量、条件变量等）设计读写锁算法，编写 pthread 程序实现你的读写锁算法。
2. 设计测试程序：测试读写锁用于共享数据结构（如链表）的正确性；
3. 测试读写锁性能，与链表整体加 mutex 保护等方式进行性能对比。

二、 算法设计和实现

1. 用条件变量实现读写锁

（1） 实现思路

读写锁是一种特殊的锁，对资源的访问者划分成读者和写者，读者只对共享资源进行读操作，写者则需要对共享资源进行写操作。一次只有一个线程可以占有写模式的读写锁，但是可以有多个线程占有读模式的读写锁。

写加锁状态：在此锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞；

读加锁状态：所有试图以读模式对它进行加锁的线程都可以获得锁，但是所有试图以写模式对这个锁加锁的线程都会被阻塞；

读优先：给读者优先权，只要读写锁不是写加锁的状态，就可以进行读加锁，同时，当一个写线程解锁时，它应该优先唤醒所有的读加锁等待的线程；

写优先：给写者优先权，只有在当前状态不是写加锁并且没有写等待时，读加锁才能成功，否则就要等待，同时，当一个写线程解锁时，它应该优先唤醒一个写加锁等待的线程。

使用条件变量和互斥量实现读写锁。本实验中使用写优先的方式进行处理。

（2） 伪代码

1. 数据结构

```
struct my_rwlock_t {
    pthread_mutex_t mutex; //互斥量
    pthread_cond_t    read; //条件读锁
    pthread_cond_t    write; //条件写锁

    //读写相关变量，正在读/写的线程数，等待读/写的线程数
    int read_now, read_wait, write_now, write_wait;
};
```

2. 主要函数

（1） 初始化条件读写锁(初始化互斥量，读写锁和相关变量)

```
void my_rwlock_init(my_rwlock_t* rwlock)
{
    pthread_mutex_init(&(rwlock->mutex), NULL);
    pthread_cond_init(&(rwlock->read), NULL);
    pthread_cond_init(&(rwlock->write), NULL);
    初始化 read_now,read_wait,write_now,write_wait;
}
```

（2） 读加锁

```
void my_rwlock_rdlock(my_rwlock_t* rwlock)
{
   pthread_mutex_lock(&(rwlock->mutex)); //互斥量加锁
    //如果有等待写或者正在写的线程不能加锁，读等待加 1
    if (rwlock->write_wait > 0 || rwlock->write_now > 0)
          read_wait+1
          // 等待被唤醒
          pthread_cond_wait
          //唤醒后，读等待线程数减 1，正在读线程数加 1
          read_wait – 1, read_now + 1
      //否则 可以直接读，正在读线程数加 1
     else read_now + 1;
  pthread_mutex_unlock(&(rwlock->mutex)); ////互斥量解锁
}
```

（3） 写加锁

```
void my_rwlock_wrlock(my_rwlock_t* rwlock) {
      pthread_mutex_lock(&(rwlock->mutex)); //互斥量加锁
     //如果没有读没有写情况下，可以写加锁，正在写加 1
     if (rwlock->read_now == 0 && rwlock->write_now == 0)
          write_now + 1;
```

```
        //否则写等待，等待被唤醒，
        else write_wait + 1 ,pthread_cond_wait
            //被唤醒后，写等待线程减 1，正在写线程加 1
              write_wait - 1; write_now + 1
        pthread_mutex_unlock(&(rwlock->mutex)); //互斥量解锁
      }
```

(4) 解锁

```
void my_rwlock_unlock(my_rwlock_t* rwlock) {
       pthread_mutex_lock(&(rwlock->mutex)); //互斥量加锁
       //读锁：当前读的线程数大于1，减1
         if (rwlock->read_now > 1) read_now - 1;
        //如果只有一个在读，唤醒写等待（如果有）
        if (rwlock->read_now == 1) read_now – 1
             if (rwlock->write_wait > 0)
                  pthread_cond_signal(&(rwlock->write));
     //写锁：正在写的线程减1
          write_now - 1;
          //写优先，唤醒写等待（如果有），没有唤醒所有读等待
           if (rwlock->write_wait > 0)
               pthread_cond_signal(&rwlock->write);
          else if (rwlock->read_wait > 0)
               pthread_cond_broadcast(&(rwlock->read));
       pthread_mutex_unlock(&(rwlock->mutex)); //互斥量解锁
};
```

3. 用信号量实现读写锁

(1) int sem_wait(sem_t *sem); // 信号量值减 1，若已为 0 则阻塞

(2) int sem_post(sem_t *sem); // +1，若原来为 0 则可能唤醒阻塞线程

(3) int sem_destroy(sem_t *sem); //释放信号量：

写模式

```
sem_wait(&w_sem);
write_operation
sem_post(&w_sem);
```

读模式

```
sem_wait(&r_sem);
if(readers == 0)
     sem_wait(&w_sem);
readers++;
sem_post(&r_sem);
read_operation
sem_wait(&r_sem);
readers- -;
```

```
if(readers == 0)
      sem_post(&w_sem);
sem_post(&r_sem);
```

4. 用互斥量实现读写锁（做性能对比）

(1) 读/写加锁 pthread_mutex_lock(&mutex);

(2) 解锁 pthread_mutex_unlock(&mutex);

写模式

```
pthread_mutex_lock(&w_mutex);
write_operation
pthread_mutex_unlock(&w_mutex);
```

读模式

```
pthread_mutex_lock(&r_mutex);
if(readers == 0)
      pthread_mutex_lock(&w_mutex);
readers++;
pthread_mutex_unlock(&r_mutex);
read_operation
pthread_mutex_lock(&r_mutex);
readers- -;
if(reader == 0)
      pthread_mutex_unlock(&w_mutex);
pthread_mutex_unlock(&r_mutex);
```

5. 其他函数

（1） list_node_s *Creatlist(int n)//创建链表

（2） int Member(int value, struct lisd_noed_s * head_p)//链表查询->读

（3） int Insert(int value, struct list_node_s** head_p)//链表插入->写

（4） int Delete(int value, struct list_node_s** head_p)//链表删除->写

（5） void Outlink(list_node_s *head) //输出链表

（6） void *ListOperation1(void *parm)//每个线程互斥量读写锁

void *ListOperation2(void *parm)//条件变量读写锁

void *ListOperation3(void *parm)//信号量读写锁

三个函数的结构

```
{
  读加锁-查询-解锁
  写加锁-插入/删除-解锁
}
```

三、 实验及结果分析

1. 实验说明

运行平台：Windows 10，线程数：5

测试程序：链表操作，读操作对应链表查询，写操作对应链表插入和删除。

2. 实验结果

实验设置：

（1） 链表大小为 1000；

（2） 共进行 160000 次操作；

（3） 设置两种读写比例(99.99%查询, 0.05%插入, 0.05%删除和 80%查询, 10%插入，10%删除）

（4） 表中计算的是线程平均运行时间（ms）

表 1 三种读写锁性能对比（99.99%查询，0.05%插入，0.05%删除）

List keys:1000, list operation: 160000 ops 99.9% Member, 0.05% Insert, 0.05% Delete						
Implementation	Number of threads(ms)					
	1	2	4	8	16	32
Mutex_rwlock	139.56	159.98	146.36	125.72	165.77	258.89
Conditional_rwlock	1517.64	722.28	470.02	422.86	443.29	482.58
Signal_rwlock	1319.14	922.16	1893.26	2396.63	2312.18	2297.13

表 2 三种读写锁性能对比（80%查询，10%插入，10%删除）

List keys:1000, list operation: 160000 ops 80% Member, 10% Insert, 10% Delete						
Implementation	Number of threads(ms)					
	1	2	4	8	16	32
Mutex_rwlock	227.51	349.63	325.06	446.50	282.53	427.64
Conditional_rwlock	1918.09	1188.45	1248.90	1769.60	1775.13	1564.61
Signal_rwlock	1652.35	1399.60	2076.65	2345.94	2730.81	2406.99

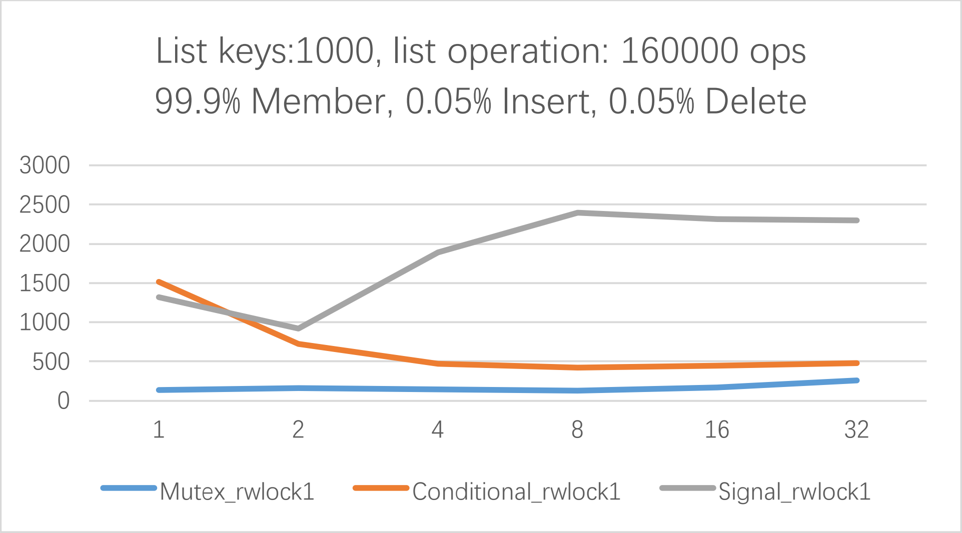


图 1 三种读写锁性能对比图（99.99%查询，0.05%插入，0.05%删除）

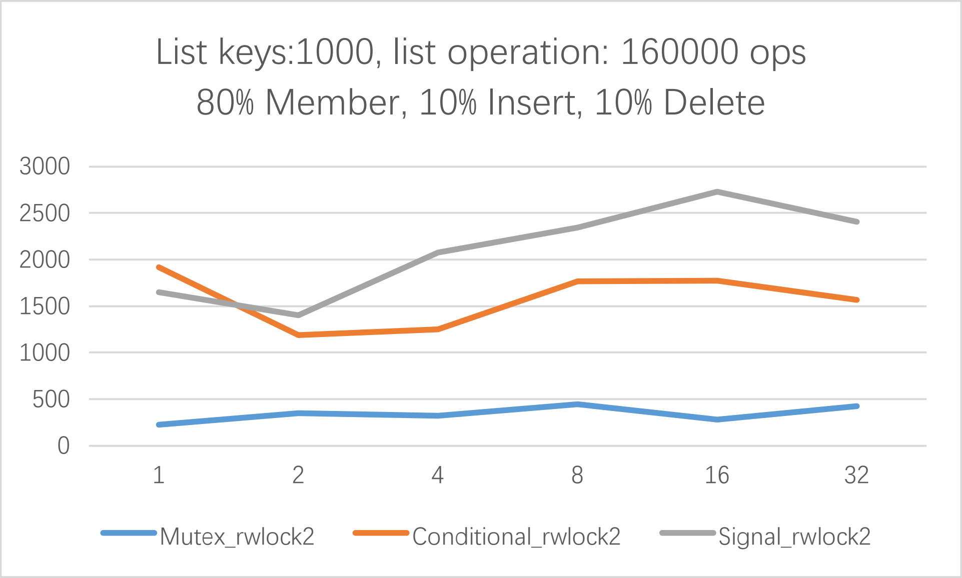


图 2 三种读写锁性能对比图（80%查询，10%插入，10%删除）

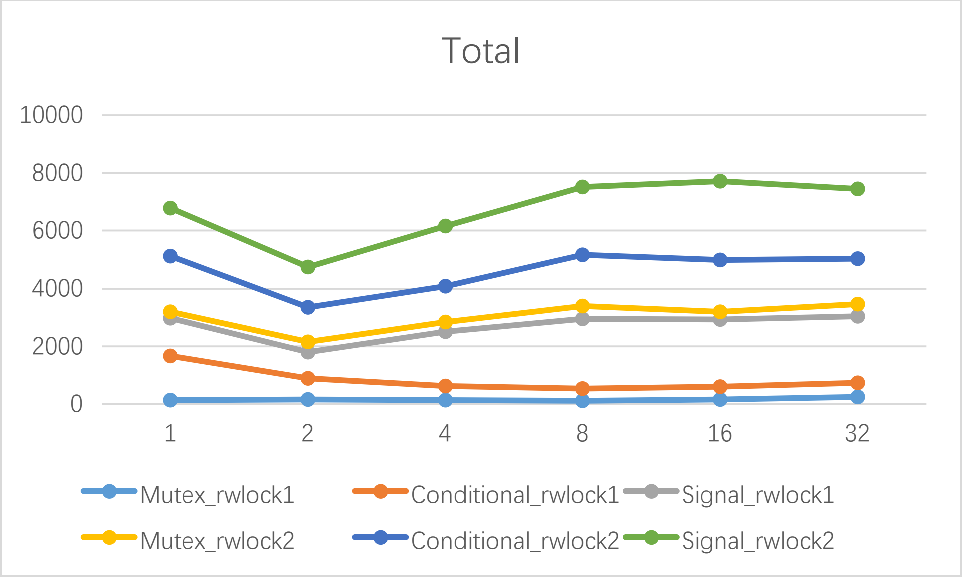


图 3 总图（1 代表 99.99%查询，0.05%插入，0.05%删除；2 代表 80%查询，10%插入，10%删除）

3. 实验分析

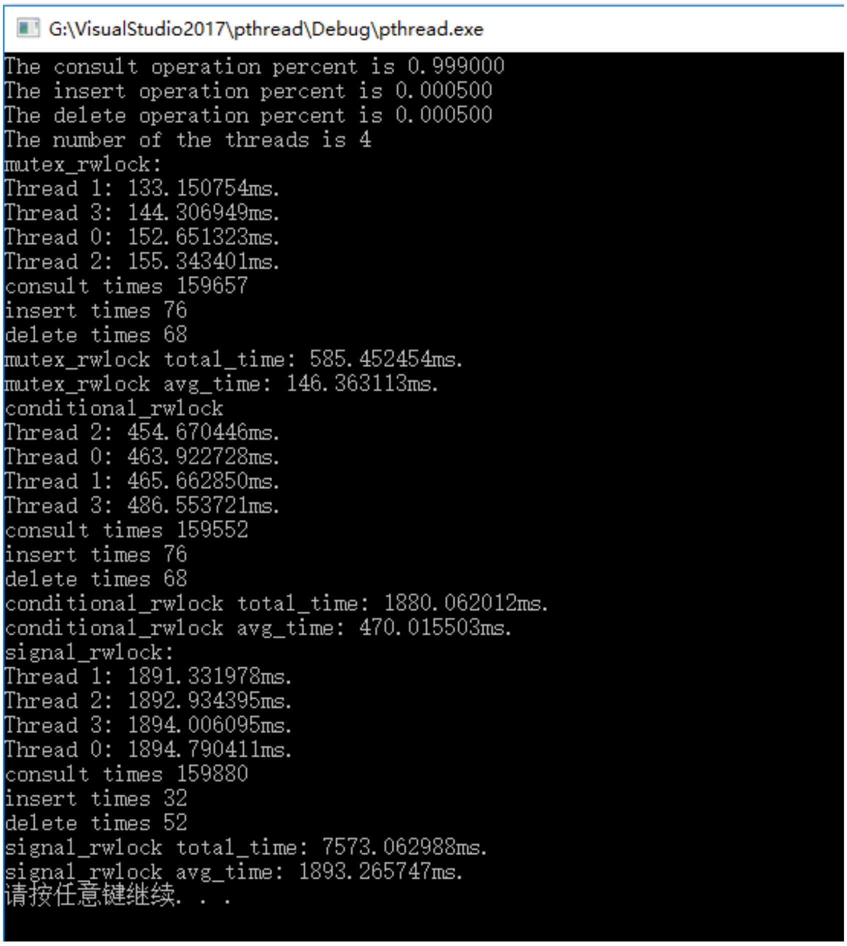
（1） 从图 3 中可以看出，当写操作的比例上升的时候，程序的运行时间显著增加。这是因为当写的比例上升的时候，写必须互斥，也就是串行执行，所有其他的读操作或者写操作都必须等待，所以程序的运行时间显著增加。

（2） 从图 1 和图 2 中可以看出，三种读写锁的性能中使用信号量实现的性能最差；mutex 的读写锁性能最好。这是由于 mutex 是 pthread 库内置的，而条件变量的读写锁用到了互斥锁，所以 mutex 性能最好，条件变量读写锁性能略差。信号量读写锁也使用了 mutex 在其基础上进行信号量判断，性能最差。

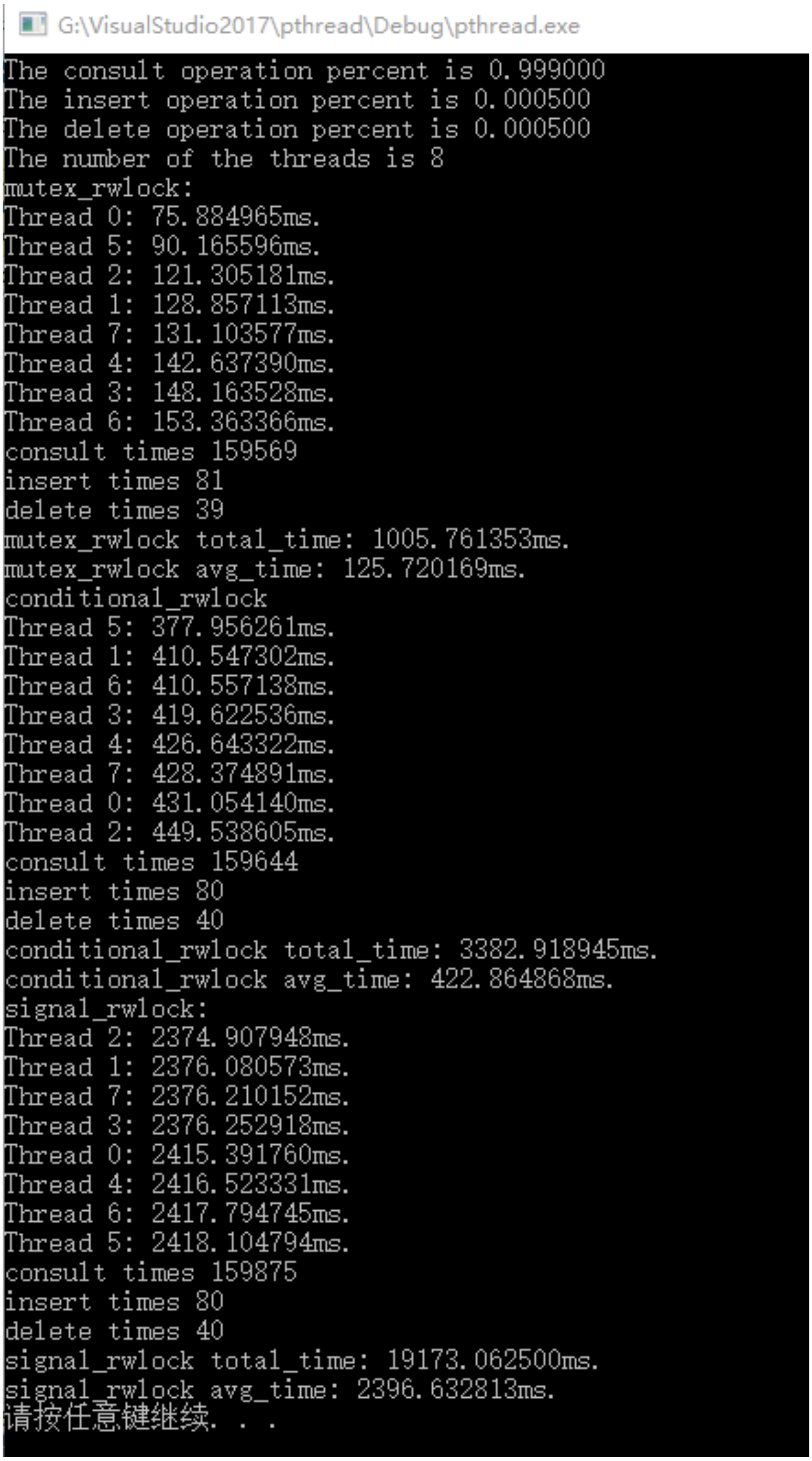
（3） 如图 1，当读操作的比例比较高的时候，在写优先条件下，使用条件变量自己实现的读写锁与 mutex 读写锁性能差别不是很大；但是当写操作比例上升的时候，如图 2， 使用条件变量自己实现的读写锁明显变慢，与 mutex 读写锁性能差别增加。在写操作比例很少的时候，写优先要求无写等待或者写加锁的时候才能进行加锁，所以写操作比例很小的时候，自己实现的条件变量读写锁进行等待的情况很少，性能于 mutex 差别不大，但当写比例上升的时候性能变差。

（4） Linux 是读优先的，所以实验可以进行扩展将读优先情况下的条件变量读写锁与 mutex 和信号量读写锁进行对比。

4. 实验部分截图



```
The consult operation percent is 0.999000
The insert operation percent is 0.000500
The delete operation percent is 0.000500
The number of the threads is 4
mutex_rwlock:
Thread 1: 133.150754ms.
Thread 3: 144.306949ms.
Thread 0: 152.651323ms.
Thread 2: 155.343401ms.
consult times 159657
insert times 76
delete times 68
mutex_rwlock total_time: 585.452454ms.
mutex_rwlock avg_time: 146.363113ms.
conditional_rwlock
Thread 2: 454.670446ms.
Thread 0: 463.922728ms.
Thread 1: 465.662850ms.
Thread 3: 486.553721ms.
consult times 159552
insert times 76
delete times 68
conditional_rwlock total_time: 1880.062012ms.
conditional_rwlock avg_time: 470.015503ms.
signal_rwlock:
Thread 1: 1891.331978ms.
Thread 2: 1892.934395ms.
Thread 3: 1894.006095ms.
Thread 0: 1894.790411ms.
consult times 159880
insert times 32
delete times 52
signal_rwlock total_time: 7573.062988ms.
signal_rwlock avg_time: 1893.265747ms.
请按任意键继续. . .
```



```
The consult operation percent is 0.999000
The insert operation percent is 0.000500
The delete operation percent is 0.000500
The number of the threads is 8
mutex_rwlock:
Thread 0: 75.884965ms.
Thread 5: 90.165596ms.
Thread 2: 121.305181ms.
Thread 1: 128.857113ms.
Thread 7: 131.103577ms.
Thread 4: 142.637390ms.
Thread 3: 148.163528ms.
Thread 6: 153.363366ms.
consult times 159569
insert times 81
delete times 39
mutex_rwlock total_time: 1005.761353ms.
mutex_rwlock avg_time: 125.720169ms.
conditional_rwlock
Thread 5: 377.956261ms.
Thread 1: 410.547302ms.
Thread 6: 410.557138ms.
Thread 3: 419.622536ms.
Thread 4: 426.643322ms.
Thread 7: 428.374891ms.
Thread 0: 431.054140ms.
Thread 2: 449.538605ms.
consult times 159644
insert times 80
delete times 40
conditional_rwlock total_time: 3382.918945ms.
conditional_rwlock avg_time: 422.864868ms.
signal_rwlock:
Thread 2: 2374.907948ms.
Thread 1: 2376.080573ms.
Thread 7: 2376.210152ms.
Thread 3: 2376.252918ms.
Thread 0: 2415.391760ms.
Thread 4: 2416.523331ms.
Thread 6: 2417.794745ms.
Thread 5: 2418.104794ms.
consult times 159875
insert times 80
delete times 40
signal_rwlock total_time: 19173.062500ms.
signal_rwlock avg_time: 2396.632813ms.
请按任意键继续. . .
```