操作系统实验报告

实验：	Lab2 Threads and Synchronization
专业：	计算机科学与技术
班级：	1 班
姓名：	姚怀聿
学号：	22920202204632

2023 年 4 月 29 日

目　　录

一、实验目的

实现 Nachos 的同步机制：锁和条件变量，并利用这些同步机制实现几个基础工具类。

二、实验要求

1. 使用 Thread::Sleep 实现锁机制和条件变量。
2. 使用 Semaphore 实现锁机制和条件变量。
3. 使用锁机制和条件变量将实验 1 里实现的双向链表修改成线程安全的，对第 1 点、第 2 点的实现应分别测试。
4. 实现一个线程安全的表结构。
5. 实现一个大小受限的缓冲区。

三、实验设计及关键代码实现

0. 实验准备

1) Mesa 语义和 Hoare 语义

- 本实验要求使用的是 Mesa 语义，这两者的区别在于：
- 假如有两个线程 T1 和 T2，T1 因为资源不满足正处于等待队列中，这时有线程释放了 T1 所需要的资源，那么这个时候会发生什么呢？T1 会立即被唤醒还是等待 T2 运行结束后再被唤醒？这种情形下的两种不同操作区分了两种语义：

    a) 对于 Mesa 语义来说，需要等到 T2 运行结束后，T1 才会被唤醒。

    b) 对于 Hoare 语义来说，T1 会被立刻唤醒。

- 简单来说，也就想当于 Mesa 是非抢占式的，而 Hoare 是抢占式的。

- 本实验中，使用 Mesa 语义，所以 signal 之后需要等待线程结束。

2) 相关的函数机制

- Thread::Sleep(): 将当前的线程设置为阻塞状态
- Scheduler::ReadyToRun(Thread* thread): 将传入函数的线程 thread 设置为就绪态后，添加到就绪队列中。
- 中断相关函数：

  在使用 Sleep 之前，为了防止错误的中断，必须暂时关闭中断功能：

  IntStatus oldLevel = interrupt->SetLevel(IntOff); 用于关闭中断。

  (void) interrupt->SetLevel(oldLevel); 用于开启中断。

3) 理解信号的 P、V 操作

P 和 V 函数实现的内容和课本上的 semWait()和 semSignal()一样。

- 在 P 操作中，如果当前的 value == 0，说明信号量被占用，则用该线程加入到信号量的阻塞队列中，否则该线程拥有信号量，value--。
- 在 V 操作中，value++，若阻塞队列不为空则从队列中取出一个线程放入就绪队列中。

1. 使用 Thread::Sleep 实现锁机制和条件变量

1) Lock 类头文件定义

- bool state;

  锁，即互斥量，可以用来保护临界区，有两种状态：上锁和解锁，因此 Lock 类中需要有一个 bool 类型的变量 state 记录当前锁的状态。当 state 为 true 时表示锁已经被占用，否则表示锁是空闲的。

- List* queue;

如果锁处于解锁状态，那么当一个线程需要请求锁的时候，该线程可以得到锁，否则当锁被占用的时候，申请该锁的线程就会被阻塞，为了记录由于等待该锁而被阻塞的线程，Lock 类中需要有一个 List 的链表存放阻塞的线程。

- Thread* threadHoldLock;

只有拥有锁的线程才能释放锁，因此类中需要有 Thread 变量来记录当前拥有锁的是哪一个线程。

```
class Lock {

  public:

    Lock(char* debugName);

    ~Lock();

    char* getName() { return name; }

    void Acquire();

    void Release();

    bool isHeldByCurrentThread();

  private:

    char* name;

    bool state;

    List* queue;

    Thread* threadHoldLock;
};
```

2) Lock 类成员函数实现

- void Lock::Acquire();

  线程尝试获取锁：首先需要关闭中断，如果锁的状态是 true，表示它已经被其他线程获取了，则将该线程添加到该锁的阻塞队列上；否则修改锁的状态、记录获得该锁的线程，然后开启中断功能。

- void Lock::Release();

  锁被释放：和 Acquire 一样，在执行操作之前同样需要先关闭中断。当锁的阻塞队列非空时，将一个线程从阻塞队列中移出，将这个线程的状态改为就绪态后，放入就绪队列中，同时修改锁的状态为 false，最后开启中断。

- bool Lock::isHeldByCurrentThread();

  判断该锁是否被当前线程所拥有，只需要判断 currentThread 和拥有该锁的线程是否相同即可。

```
Lock::Lock(char* debugName) {

    name = debugName;

    queue = new List();

}

Lock::~Lock() {

    delete queue;

}
```

```
void Lock::Acquire() {

        IntStatus oldLevel = interrupt->SetLevel(IntOff);

        if(state == true) {

                /*block the current thread*/

                queue->Append((void*)currentThread);

                currentThread->Sleep();

        }

        state = true; // lock the lock

        threadHoldLock = currentThread;

        (void*) interrupt->SetLevel(oldLevel);

}

void Lock::Release() {

        IntStatus oldLevel = interrupt->SetLevel(IntOff);

        Thread* thread;

        thread = (Thread*)queue->Remove(); // get the first item and let it be ready

        if(thread != NULL) {

                scheduler->ReadyToRun(thread);

        }
        state = false;

        threadHoldLock = NULL;

        (void*) interrupt->SetLevel(oldLevel);

}

bool Lock::isHeldByCurrentThread() {

        return currentThread == threadHoldLock;

}
```

3) Condition 类头文件定义

- List* queue;

  条件变量是利用线程间共享的全局变量进行同步的一种机制，当条件变量的条件不成立时，线程需要被阻塞挂起，当等待的条件成立时，线程被重新唤醒，放入就绪队列中，因此需要一个队列来存放阻塞的线程。

```
class Condition {

    public:

        Condition(char* debugName);

        ~Condition();

        char* getName() { return (name); }

        void Wait(Lock *conditionLock);

        void Signal(Lock *conditionLock);
        void Broadcast(Lock *conditionLock);

    private:

        char* name;

        List* queue;

};
```

4) Condition 类成员函数实现

- void Condition::Wait(Lock *conditionLock);

  该函数等待条件变量的条件成立，当条件不成立时，使用该函数让线程等待条件成立。同样需要先关闭中断，防止发生意想不到的错误，首先确保当前线程拥有该锁，否则可能造成临界资源的访问错误，将该线程添加到阻塞挂起队列中，然后释放锁，将当前线程的状态切换到阻塞态后，等待条件成立被唤醒而重新得到锁。

- void Condition::Signal(Lock *conditionLock);

  在确保当前线程拥有锁的情况下，如果因条件变量而阻塞的线程不空，则从队列中唤醒一个线程，并将该线程添加到就绪队列中。

- void Condition::Broadcast(Lock *conditionLock);

```
Condition::Condition(char* debugName) {

    name = debugName;

    queue = new List();

}
```

```
Condition::~Condition() {

        delete queue;

}

void Condition::Wait(Lock* conditionLock) {

        IntStatus oldLevel = interrupt->SetLevel(IntOff);

        ASSERT(conditionLock->isHeldByCurrentThread()); // make sure that the
current thread hold the lock

        queue->Append((void*)currentThread);

        conditionLock->Release(); // release before change status

        currentThread->Sleep();

        conditionLock->Acquire(); // get the lock after change

        (void*) interrupt->SetLevel(oldLevel);

}

void Condition::Signal(Lock* conditionLock) { // wakeup a thread in the blocked
queue and let it added to the ready queue

        IntStatus oldLevel = interrupt->SetLevel(IntOff);

        Thread *thread;

        thread = (Thread*)queue->Remove();

        if(thread != NULL) {
```

```
        scheduler->ReadyToRun(thread);
    }
    (void*) interrupt->SetLevel(oldLevel);
}
void Condition::Broadcast(Lock* conditionLock) { // wakeup all threads that are
blocked
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    Thread *thread = (Thread*)queue->Remove();
    while(thread != NULL) {
        scheduler->ReadyToRun(thread);
        thread = (Thread*)queue->Remove();
    }
    (void*) interrupt->SetLevel(oldLevel);
}
```

5) 测试该锁是否正确实现

- 创建一个测试函数 SynchTest，其功能为：

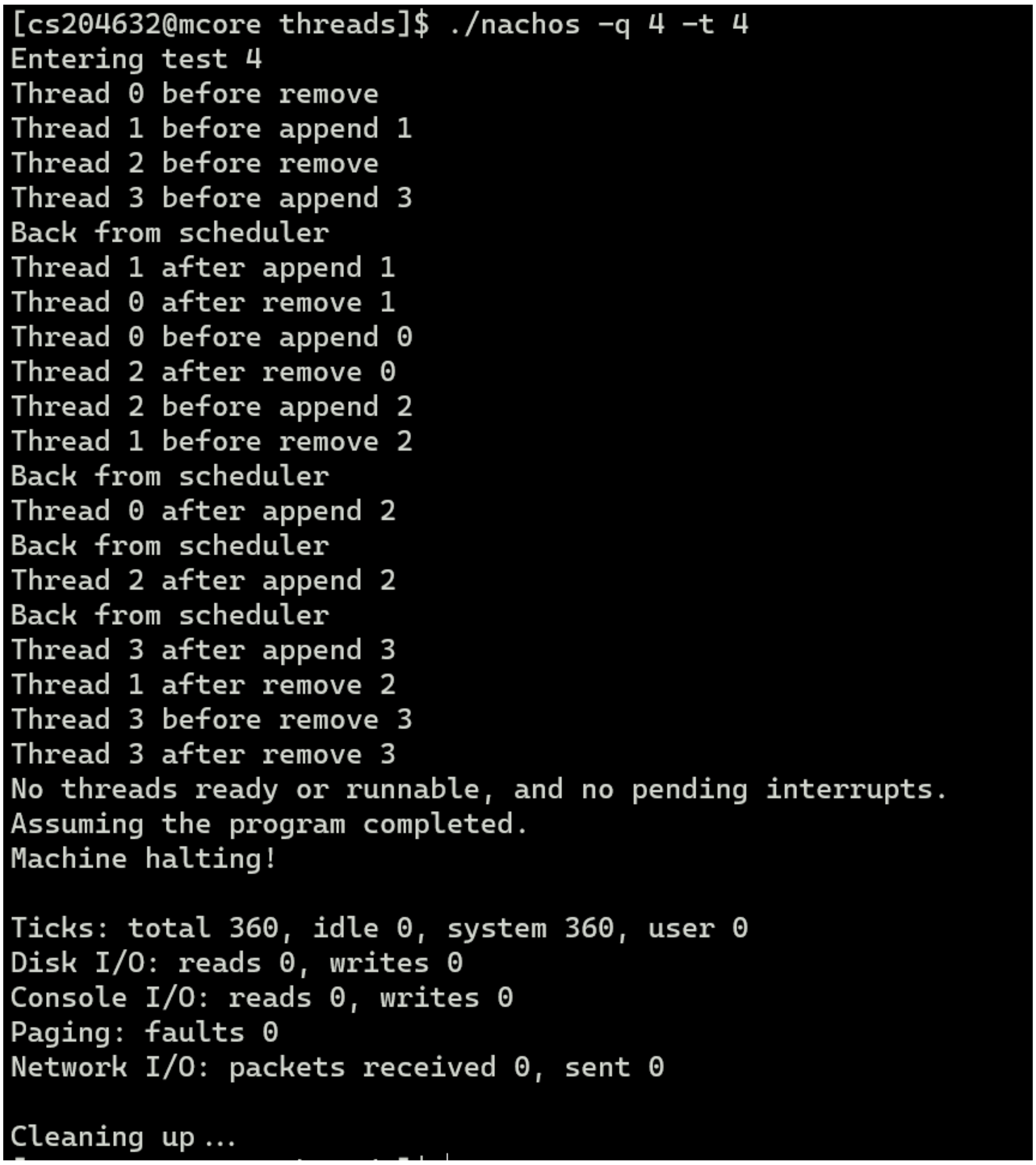
  当线程编号为偶数时，先从 SynchList 中移出元素，再向 SynchList 中添加元素；当线程编号为奇数时，先向 SynchList 中添加元素，再从 SynchList 中移出元素：

```
void SyncTest(int which) {

    if(which % 2 == 0) { // even first remove then add

        fprintf(stdout, "Thread %d before remove\n", which);

        int* item = (int*)slist->Remove();

        fprintf(stdout, "Thread %d after remove %d\n", which, *item);

        *item = which;

        fprintf(stdout, "Thread %d before append %d\n", which, *item);

        slist->Append((void*)item);

        fprintf(stdout, "Thread %d after append %d\n", which, *item);

    } else { // odd first add then remove

        int* item = new int();

        *item = which;

        /* Append */

        fprintf(stdout, "Thread %d before append %d\n", which, *item);

        slist->Append((void*)item);

        fprintf(stdout, "Thread %d after append %d\n", which, *item);

        currentThread->Yield();

        /* Remove */
        fprintf(stdout, "Thread %d before remove %d\n", which, *item);

        slist->Remove();

        fprintf(stdout, "Thread %d after remove %d\n", which, *item);

        delete item;

    }

}
```

● 测试方式：在锁定临界资源的时候，切换进程，以此来验证锁的正确性。

```
void

SynchList::Append(void *item)

{

    lock->Acquire();            // enforce mutual exclusive access to the list

    list->Append(item);

    currentThread->Yield();

    fprintf(stdout, "Back from scheduler\n");

    listEmpty->Signal(lock); // wake up a waiter, if any

    lock->Release();

}
```

● 编译运行：编译成功后，在终端输入./nachos -q 4 -t 4，-q 4 表示进入第四个 test 函数（即 SynchTest），-t 4 表示使用 4 个线程。



```
[cs204632@mcore threads]$ ./nachos -q 4 -t 4
Entering test 4
Thread 0 before remove
Thread 1 before append 1
Thread 2 before remove
Thread 3 before append 3
Back from scheduler
Thread 1 after append 1
Thread 0 after remove 1
Thread 0 before append 0
Thread 2 after remove 0
Thread 2 before append 2
Thread 1 before remove 2
Back from scheduler
Thread 0 after append 2
Back from scheduler
Thread 2 after append 2
Back from scheduler
Thread 3 after append 3
Thread 1 after remove 2
Thread 3 before remove 3
Thread 3 after remove 3
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 360, idle 0, system 360, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

- 分析：

锁的有效性：从上图中可以看到，在“Thread 1 before append 1”到“Thread 1 after append 1”的过程中，线程 2、线程 3 在锁的作用下均被阻塞，最后回到线程 1 执行，输出“Back from scheduler”，说明了锁的有效性。

条件变量：在线程 1 结束“Thread 1 append 1”后，成功唤醒线程 0 删除“Thread 0 after remove 1”（线程 0 最先执行，因为一开始链表无元素，而线程 0 想执行删除操作，故被阻塞）。

2. 使用 Semaphore 实现锁机制和条件变量

使用信号量 Semaphore 实现锁机制和条件变量与使用 Thread::Sleep 实现的思路大致相同，只是需要把 Sleep 操作和 ReadyToRun 操作换为信号量的 P 和 V 操作。

1) Lock 类头文件定义

Lock 类中需要的锁状态可以用信号量中的 value 来代替，阻塞队列可以用信号量中的 queue 代替，需要添加的是**拥有锁的线程和信号量 sema**。

```
class Lock {

   public:

      Lock(char* debugName);             // initialize lock to be FREE

      ~Lock();                          // deallocate lock

      char* getName() { return name; } // debugging assist

      void Acquire(); // these are the only operations on a lock

      void Release(); // they are both *atomic*

      bool isHeldByCurrentThread();

   private:

      char* name;

      Semaphore* Sema; // use the semaphore to realize the lock

      Thread* threadHoldLock; // the thread that hold the lock, only the thread can
release the lock
};
```

2) Lock 类成员函数实现

- void Lock::Acquire();

  由于 Semaphore 的 P 操作是原子操作，所以不必考虑中断的影响，线程使用 Acquire 申请锁时，调用 Sema->P()操作，如果 value > 0，说明锁没有被占用，则修改 threadHoldLock 的值，将其改为申请的进程，否则将该进程添加到阻塞队列并 Sleep。

- void Lock::Release();

  Semaphore 的 V 操作也是原子操作，先将 threadHoldLock 的值置为 NULL，然后调用 Sema->V()操作即可。

- bool Lock::isHeldByCurrentThread();

  判断该锁是否被当前线程所拥有，只需要判断 currentThread 和拥有该锁的线程是否相同即可。

```
Lock::Lock(char* debugName) {

      name = debugName;

      Sema = new Semaphore(debugName, 1);

}

Lock::~Lock() {

      delete Sema;

}
```

```
void Lock::Acquire() {

        Sema->P();

        threadHoldLock = currentThread;

}

void Lock::Release() {

        threadHoldLock = NULL;

        Sema->V();

}

bool Lock::isHeldByCurrentThread() { //    return true if the current thread hold the
lock

        return currentThread == threadHoldLock;

}
```

3) Condition 类头文件定义

- List* queue;

  条件变量是利用线程间共享的全局变量进行同步的一种机制，当条件变量的条件不成立时，线程需要被阻塞挂起，当等待的条件成立时，线程被重新唤醒，放入就绪队列中，因此需要一个队列来存放阻塞的线程。

```
class Condition {

    public:
```

```
    Condition(char* debugName);

    ~Condition();

    char* getName() { return (name); }

    void Wait(Lock *conditionLock);

    void Signal(Lock *conditionLock);      // conditionLock must be held by

    void Broadcast(Lock *conditionLock);// the currentThread for all of

  private:

    char* name;

    Semaphore* Sema;

    int blockNum;

};
```

4) Condition 类成员函数实现

- void Condition::Wait(Lock *conditionLock);

  首先释放锁，调用 Sema->P()函数将进程设置为阻塞态并放入阻塞队列中，同时阻塞线程数加 1，最后等待条件成立，重新被唤醒。

- void Condition::Signal(Lock *conditionLock);

  如果当前被阻塞的线程数大于 0，则调用 Sema->V()释放一个线程，并将其加入到就绪队列中去。

- void Condition::Broadcast(Lock *conditionLock);

当阻塞线程数大于 0 时，重复调用 Sema->V()，直到所有的阻塞线程都被重载到就绪队列中去。

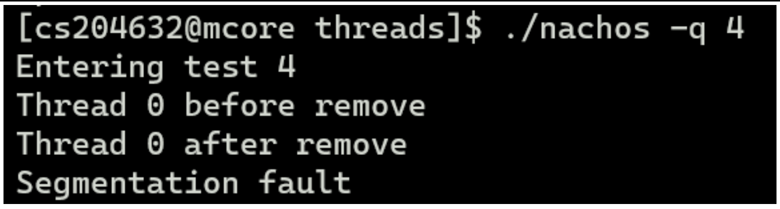
```
Condition::Condition(char* debugName) {

    name = debugName;

    Sema = new Semaphore(debugName, 0);

    blockNum = 0;

}

Condition::~Condition() {

    delete Sema;

}

void Condition::Wait(Lock* conditionLock) {

    conditionLock->Release();

    Sema->P();

    blockNum++;

    conditionLock->Acquire();

}

void Condition::Signal(Lock* conditionLock) { // wakeup a thread in the blocked
queue and let it added to the ready queue

    if(blockNum > 0) {
            Sema->V();

            blockNum--;

        }

}

void Condition::Broadcast(Lock* conditionLock) { // wakeup all threads that are
blocked

        while(blocksum) {

            Sema->V();

            blockNum--;

        }

}
```

3. 使用锁机制和条件变量将实验 1 里实现的双向链表修改成线程安全的

1) 测试以“sleep+中断禁止与启用”实现的条件变量是否正确

- 测试代码：奇数编号线程先添加后删除，偶数编号线程先删除后添加。

```
void DLListTest(int which) {
        if(which % 2 == 0) { // even first remove then add
            fprintf(stdout, "Thread %d before remove\n", which);
            dllist->Remove(NULL);
            fprintf(stdout, "Thread %d after remove\n", which);
            fprintf(stdout, "Thread %d before append\n", which);
            dllist->Append(NULL);
            fprintf(stdout, "Thread %d after append\n", which);
        } else { // odd first add then remove
            /* Append */
            fprintf(stdout, "Thread %d before append\n", which);
            dllist->Append(NULL);
            fprintf(stdout, "Thread %d after append\n", which);
            currentThread->Yield();
            /* Remove */
            fprintf(stdout, "Thread %d before remove\n", which);
            dllist->Remove(NULL);
            fprintf(stdout, "Thread %d after remove\n", which);
        }
}
```



```
[cs204632@mcore threads]$ ./nachos -q 4
Entering test 4
Thread 0 before remove
Thread 0 after remove
Segmentation fault
```

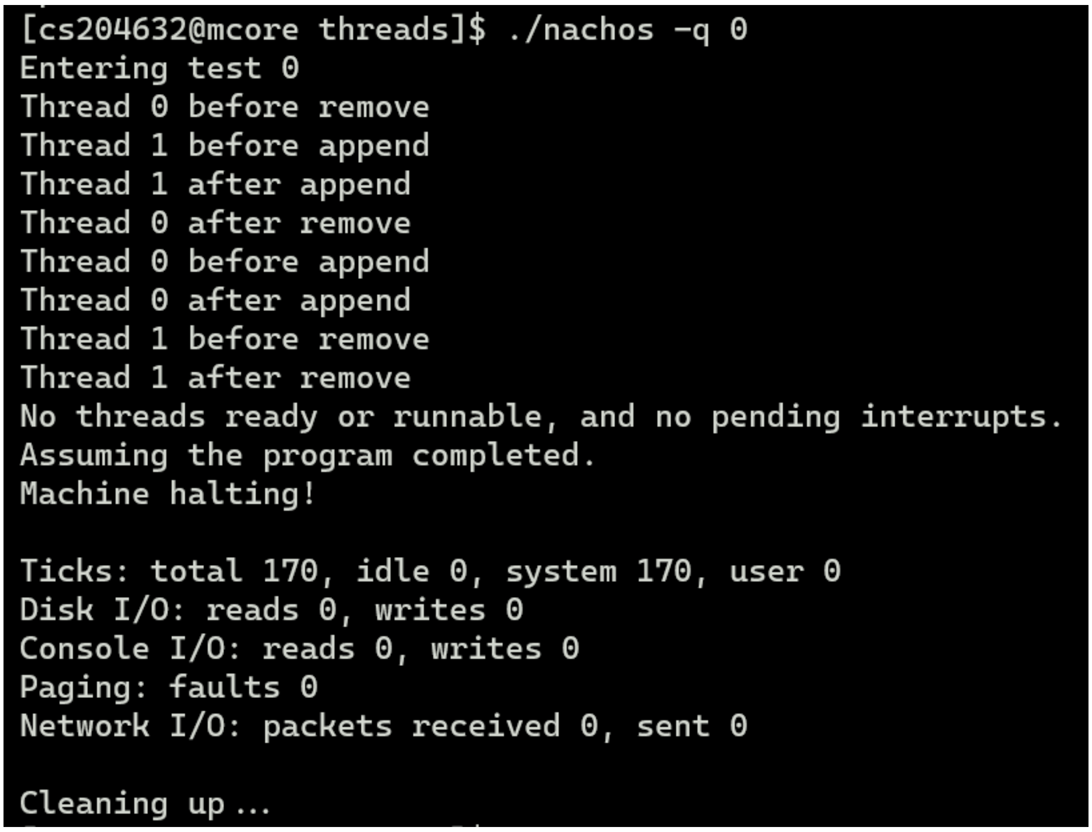
一开始线程 0 进入，先删除，但是此时链表中一个元素也没有，于是发生了段错误。

- 加入条件变量：

```
void DLList::Append(void *item) {
        lock->Acquire();
        DLLElement *newnode = new DLLElement(item, 1);
        if(IsEmpty()) {
            first = newnode;
            last = newnode;
        } else {
            last->next = newnode;
```

```
                newnode->prev = last;
                last = newnode;
        }
        dllistEmpty->Signal(lock);
        lock->Release();
}
void *DLList::Remove(int *keyPtr) { // remove from head
        lock->Acquire();
        while(IsEmpty()) dllistEmpty->Wait(lock); // condition
        DLLElement *element = first;
        void *thing;
        thing = first->item;
        if(first == last && first->next == NULL) { // only one
                first = NULL;
                last = NULL;
        } else {
                first = element->next;
        }
        if(keyPtr != NULL) *keyPtr = element->key;
        delete element;
        lock->Release();
        return thing;
}
```

再次测试：

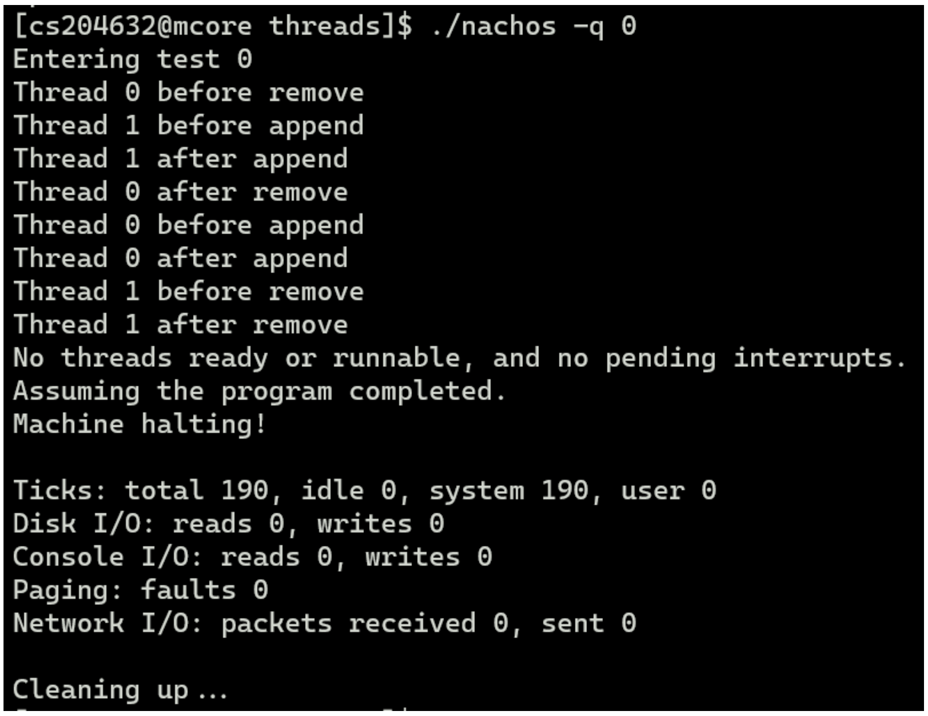


```
[cs204632@mcore threads]$ ./nachos -q 0
Entering test 0
Thread 0 before remove
Thread 1 before append
Thread 1 after append
Thread 0 after remove
Thread 0 before append
Thread 0 after append
Thread 1 before remove
Thread 1 after remove
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 170, idle 0, system 170, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看到，线程 0 进入，链表中没有元素，于是被条件变量阻塞，等待条件成立，之后线程 1 向链表中加入元素，唤醒线程 0，线程 0 删除元素，此后程序正确执行。

2) 测试以“信号量”实现的锁机制和条件变量是否正确

测试方法和前面测试“Sleep”实现的相同。下面直接给出测试结果。



```
[cs204632@mcore threads]$ ./nachos -q 0
Entering test 0
Thread 0 before remove
Thread 1 before append
Thread 1 after append
Thread 0 after remove
Thread 0 before append
Thread 0 after append
Thread 1 before remove
Thread 1 after remove
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 190, idle 0, system 190, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看到，程序同样正确执行。

至此，锁和条件变量的正确性测试完毕，且用锁和条件变量实现了双向链表的线程安全性。

4. 实现一个线程安全的表结构

1) Table 类头文件定义

- void** table; 定义表头指针，线性表的主体是一个指针数组。
- int tsize; 记录表的大小。
- int cnt; 记录表中的元素个数。
- Lock* lock; 锁。
- Condition* tableEmpty; 用来阻塞表为空的情况。

这里锁和条件变量的实现使用的是第二部分信号量的方法。

table.h

```
#ifndef __TABLE_H

#define __TABLE_H

#include "synch.h"

class Table {

public:

        Table(int size);

        ~Table();


        int Alloc(void *object);

        void Release(int index);

        void *Get(int index);

        void Print(); // print all item in tableBase

        int getSize(); // get the current table size

        int cnt;

private:
        int TableSize;

        void** tableBase;

        Lock* lock;

        Condition* tableEmpty;

};


#endif // __TABLE_H
```

2) Table 类中主要的成员函数

- 在 Table 类中最主要的两个函数是 Alloc 和 Release 函数，它们的作用分别是在表中添加元素和删除元素。

- int Alloc(void *object);

  操作前先请求锁，然后遍历线性表直到找到线性表中第一个空位的索引 index，若 index 超过了表界限则返回-1，否则将 object 添加到此位置。添加完毕后 Signal，唤醒由于线性表中 index 位置为空而被阻塞的线程，函数的返回值为插入位置的索引 index。

- void Release(int index);

  对于 Release 函数，在请求锁后首先需要判断线性表当前位置是否为空，如果为空，则线程被 tableEmpty 阻塞，直到另一个线程向线性表中该位置添加元素后重新被唤醒。

table.cc

```
#include "table.h"

#include "synch.h"


Table::Table(int size) {

    TableSize = size;

    cnt = 0; // initailize

    lock = new Lock("table lock");

    tableEmpty = new Condition("tableEmpty condition");

    tableBase = new void*[size];
    for(int i = 0; i < size; i++) {
        tableBase[i] = NULL;
    }
}

Table::~Table() {
    TableSize = 0;
    cnt = 0;
    delete tableBase;
    delete lock;
    delete tableEmpty;
}

int Table::Alloc(void *object) { // add object to tableBase
    lock->Acquire();
    int ret;
    int index = 0;
    while(tableBase[index] != NULL && index < TableSize) {
        index++;
    }
    if(index >= TableSize) {
        ret = -1;
    } else {
        tableBase[index] = object;
        ret = index;
    }
    ASSERT(ret != -1);
    cnt += 1;
```

```
	tableEmpty->Signal(lock);
	lock->Release();
	return ret;
}

void Table::Release(int index) { // release item from tableBase
	lock->Acquire();
	while(tableBase[index] == NULL) tableEmpty->Wait(lock);
	tableBase[index] = NULL;
	cnt -= 1;
	lock->Release();
}

void *Table::Get(int index) { // return item on tableBase[index]
	lock->Acquire();
	void *ret;
	if(index >= TableSize) {
		ret = NULL;
	} else {
		ret = tableBase[index];
	}
	lock->Release();
	return ret;
}

int Table::getSize() {
	return TableSize;
}
```

```
void Table::Print() {
    for(int i = 0; i < TableSize; i++) {
        if(tableBase[i] == NULL) printf("tableBase[%d] : 0\n", i);
        else printf("tableBase[%d] : %d\n", i, (int)tableBase[i]);
    }
}
```

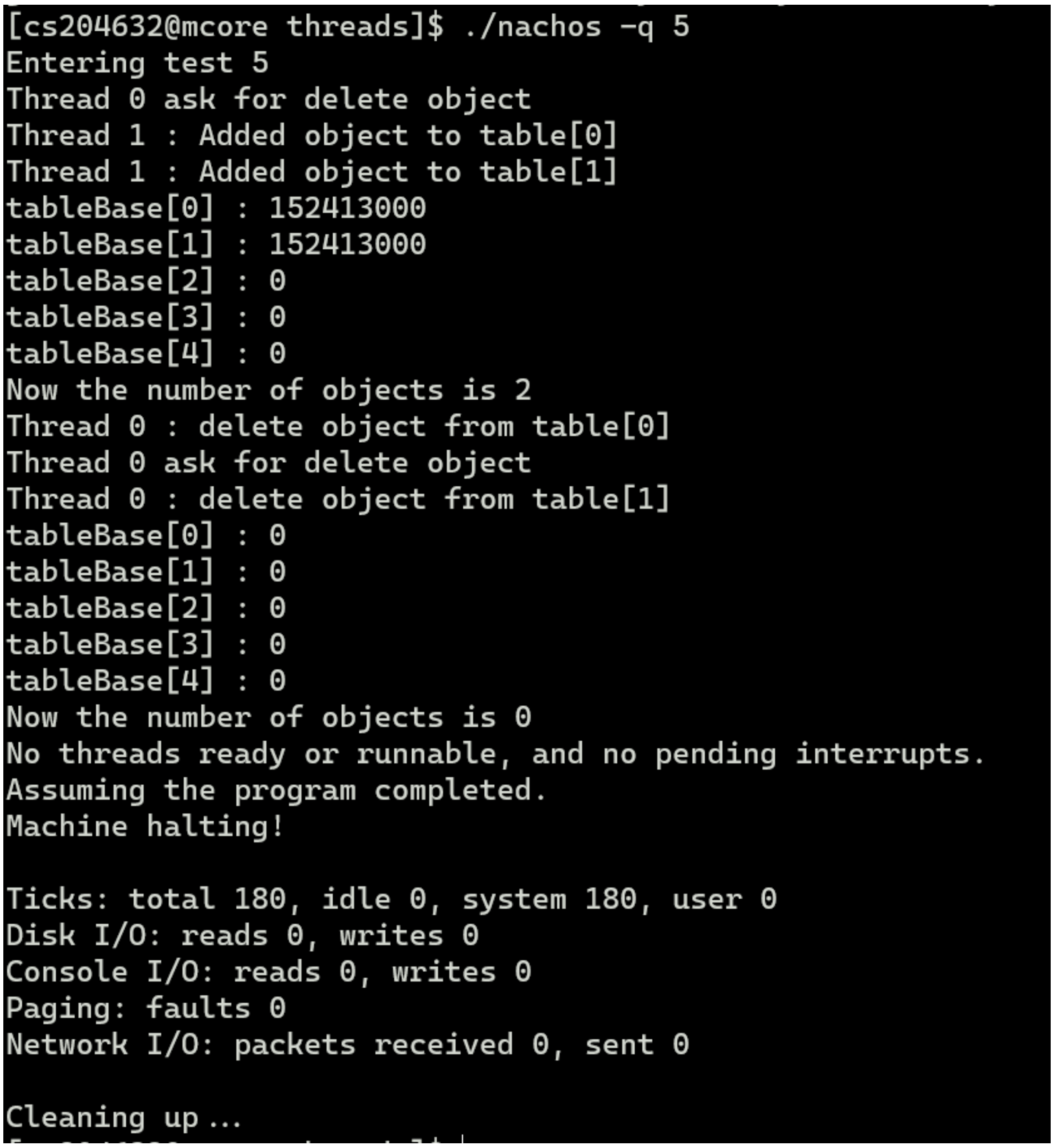
3) 测试代码

为了检测表的安全性，让 Thread0 先对表进行删除操作，Thread1 进行增加操作，测试前使用 ASSERT 函数确保操作数 oprnum 不大于表的大小。

```
void TableTest(int which) {
    for(int i = 0; i < threadnum; i++) testData[i] = i;
    ASSERT(oprnum <= table->getSize());
    int *object = new int();
    int index = 0;
    if(which % 2) {
        for(int i = 0; i < oprnum; i++) {
            *object = testData[which + i];
            index = table->Alloc((void*)object);
            printf("Thread %d : Added object to table[%d]\n", which, index);
        }
        table->Print();
        printf("Now the number of objects is %d\n", table->cnt);
    } else {
        for(int i = 0; i < oprnum; i++) {
            printf("Thread %d ask for delete object\n", which);
            table->Release(index);
            printf("Thread %d : delete object from table[%d]\n", which, index);
```

```
                index += 1;

            }

            table->Print();

            printf("Now the number of objects is %d\n", table->cnt);

        }

    }
```

4) 测试结果



```
[cs204632@mcore threads]$ ./nachos -q 5
Entering test 5
Thread 0 ask for delete object
Thread 1 : Added object to table[0]
Thread 1 : Added object to table[1]
tableBase[0] : 152413000
tableBase[1] : 152413000
tableBase[2] : 0
tableBase[3] : 0
tableBase[4] : 0
Now the number of objects is 2
Thread 0 : delete object from table[0]
Thread 0 ask for delete object
Thread 0 : delete object from table[1]
tableBase[0] : 0
tableBase[1] : 0
tableBase[2] : 0
tableBase[3] : 0
tableBase[4] : 0
Now the number of objects is 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 180, idle 0, system 180, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

从图中可以看到，实现了线程安全的表结构。

5. 实现一个大小受限的缓冲区

1) BoundedBuffer 类头文件定义

与 Table 不同，BoundedBuffer 需要两个索引 head 和 tail，head 存放的是已经使用了的 Buffer 中的第一个位置的下标，tail 存放的是已经使用的 Buffer 中的最后一个位置的下标。buffer 是缓冲区。maxSize 为 buffer 最多能存放的大小，usedSize 为当前已经使用的 buffer 的大小。

```
class BoundedBuffer {
public:
    BoundedBuffer(int maxSize);
    ~BoundedBuffer();
    void Read(void *data, int size);
    void Write(void *data, int size);
private:
    int maxSize; // size of buffer
    int head; // pointer to the first place of buffer used
    int tail; // pointer to the last place of buffer used
    int usedSize;
    uchar *buffer;
private:
    Lock *lock;
    Condition *full; // ensure buffer not full
    Condition *empty; // ensure buffer not empty
};
```

2) BoundedBuffer 类成员函数

- void BoundedBuffer::Write(void* data, int size);

  实现了向缓冲区中写入数据的功能。首先确保要写入的数据大小不大于 buffer 的最大容量。然后获取锁，保证操作的原子性，当要写的数据量超过了剩余容量，则将这个线程阻塞，使用条件变量，等待 buffer 中的数据被其他线程读取。若不超过剩余容量，则将 size 个数据依次写入 buffer 中：从 tail

= 0 开始，依次往后写，这里将 buffer 看出是一个循环队列，用取模的运算，当到达 buffer 的末尾时，就再从头写。写完后让 usedSize 增加 size 大小，然后 Signal()，唤醒由于缓冲区中元素不够而阻塞的线程，最后释放锁。

- void BoundedBuffer::Read(void* data, int size);

  实现了从缓冲区中读取数据的功能。首先获取锁，保证操作的原子性。当要从缓冲区中读取的数据个数超过了缓冲区中已使用的大小时，线程被阻塞，使用条件变量，等待其他线程将数据写入缓冲区。若不超过已用大小，则从 head 开始，将从 head 开始的后 size 个数据都放入传入的 data 变量中，之后再将 head 增加 size 个，之后再读取位置就变了，然后将已使用缓冲区的大小 usedSize 减少 size 个，最后 Signal()，唤醒由于缓冲区中元素太多而被阻塞的进程，再释放锁。

```
BoundedBuffer::BoundedBuffer(int maxSize) {
    this->maxSize = maxSize;
lock = new Lock("Buffer lock");
    full = new Condition("Buffer not full cond");
    empty = new Condition("Buffer not empty cond");
    usedSize = 0;
    head = 0;
    tail = 0;
    buffer = new uchar[maxSize];
}
BoundedBuffer::~BoundedBuffer() {
    delete lock;
    delete full;
    delete empty;
    delete buffer;
}
void BoundedBuffer::Write(void* data, int size) { // write data to buffer and the data size is size
    ASSERT(size <= maxSize);
    lock->Acquire();
    while(maxSize - usedSize < size) { // remain size not enough
        DEBUG('t', "\tRemain size is not enough.\n");
        full->Wait(lock);
    }
    for(int i = 0; i < size; i++) {
        *((uchar*)buffer + (tail + i) % maxSize) = *((uchar*)data + i);
    }
    tail = (tail + size) % maxSize;
    usedSize += size;
    DEBUG('b', "\tFinish Write Buffer. Buffer size : %d\n", usedSize);
    empty->Signal(lock);
    lock->Release();
}
void BoundedBuffer::Read(void* data, int size) {
    lock->Acquire();
    while(usedSize - size < 0) {
        DEBUG('t', "\tNot enough to read\n");
        empty->Wait(lock);
    }
    for(int i = 0; i < size; i++) {
        *((uchar*)data + i) = *((uchar*)buffer + (head + i) % maxSize);
    }
    head = (head + size) % maxSize;
    usedSize -= size;
    DEBUG('b', "\tFinish Read buffer. Buffer size : %d\n", usedSize);
    full->Signal(lock);
    lock->Release();
}
```

```
void BoundedBuffer::Print() {
    printf("\t");
    for(int i = head; i != tail; i = (i + 1) % maxSize) {
        printf("%c", buffer[i]);
    }
    printf("\n");
}
```

3) 测试函数

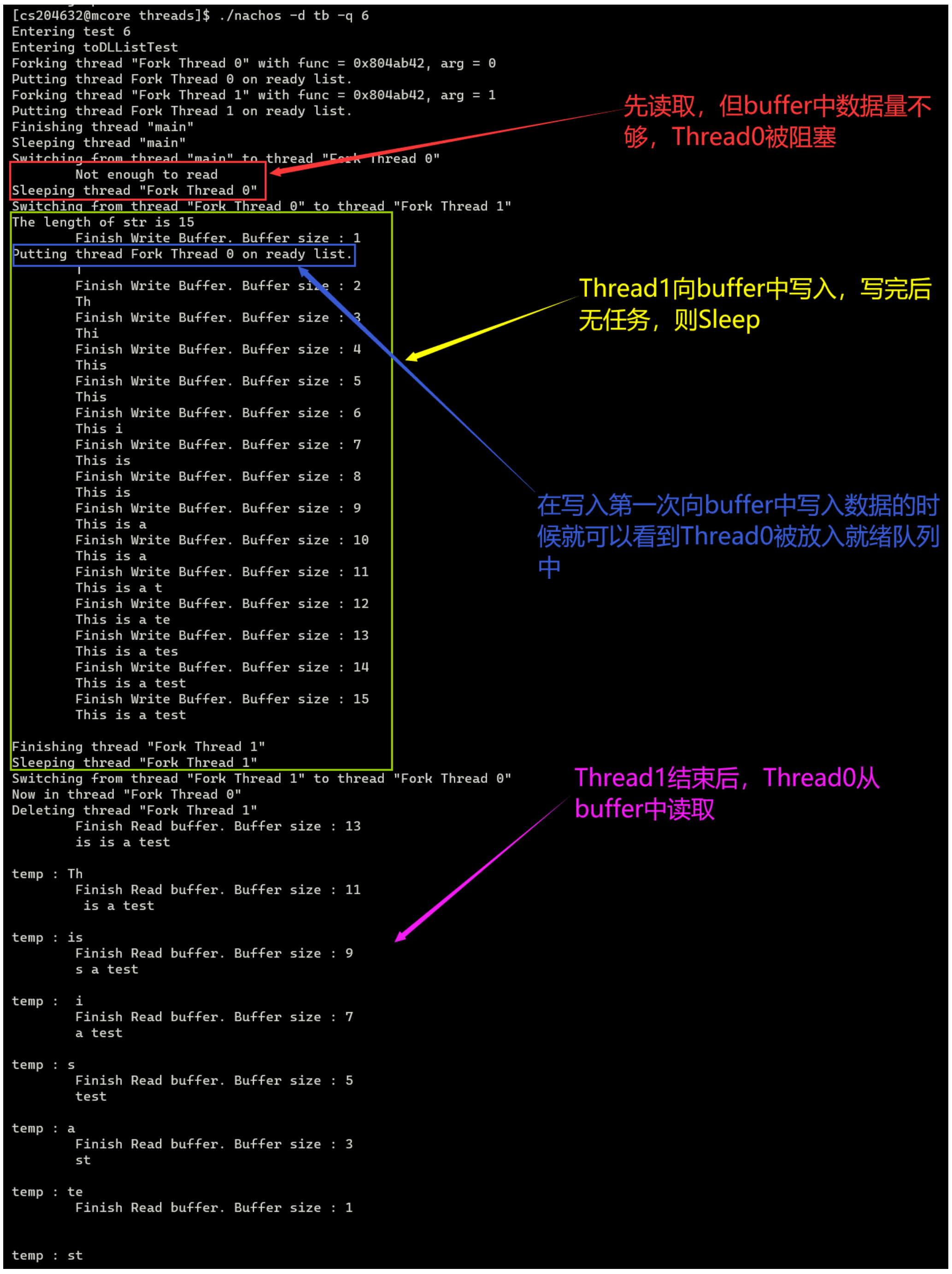
测试逻辑与 Table 测试逻辑相似，编号为偶数的线程从缓冲区中读取，而编号为奇数的线程向缓冲区中写入，这样由于先 fork 线程 0，会构成一个先读取后写入的顺序，以此来检验 buffer 的安全性。

```
void BoundedBufferTest(int which) {
    if(which % 2) {
        char *str = "This is a test\n";
        printf("The length of str is %d\n", strlen(str));
        for(int i = 0; i < 100; i++) {
            char temp = str[i];
            bb->Write(&temp, 1);
            bb->Print();
            if(temp == '\n') {
                break;
            }
        }
    } else {
        for(int i = 0; i < 100; i++) {
            char *temp = new char[3];
            temp[2] = '\0';
            bb->Read(temp, 2);
```

```
                bb->Print();

                printf("temp : %s\n", temp);

            }

        }

}
```

4) 测试结果

- 常规测试



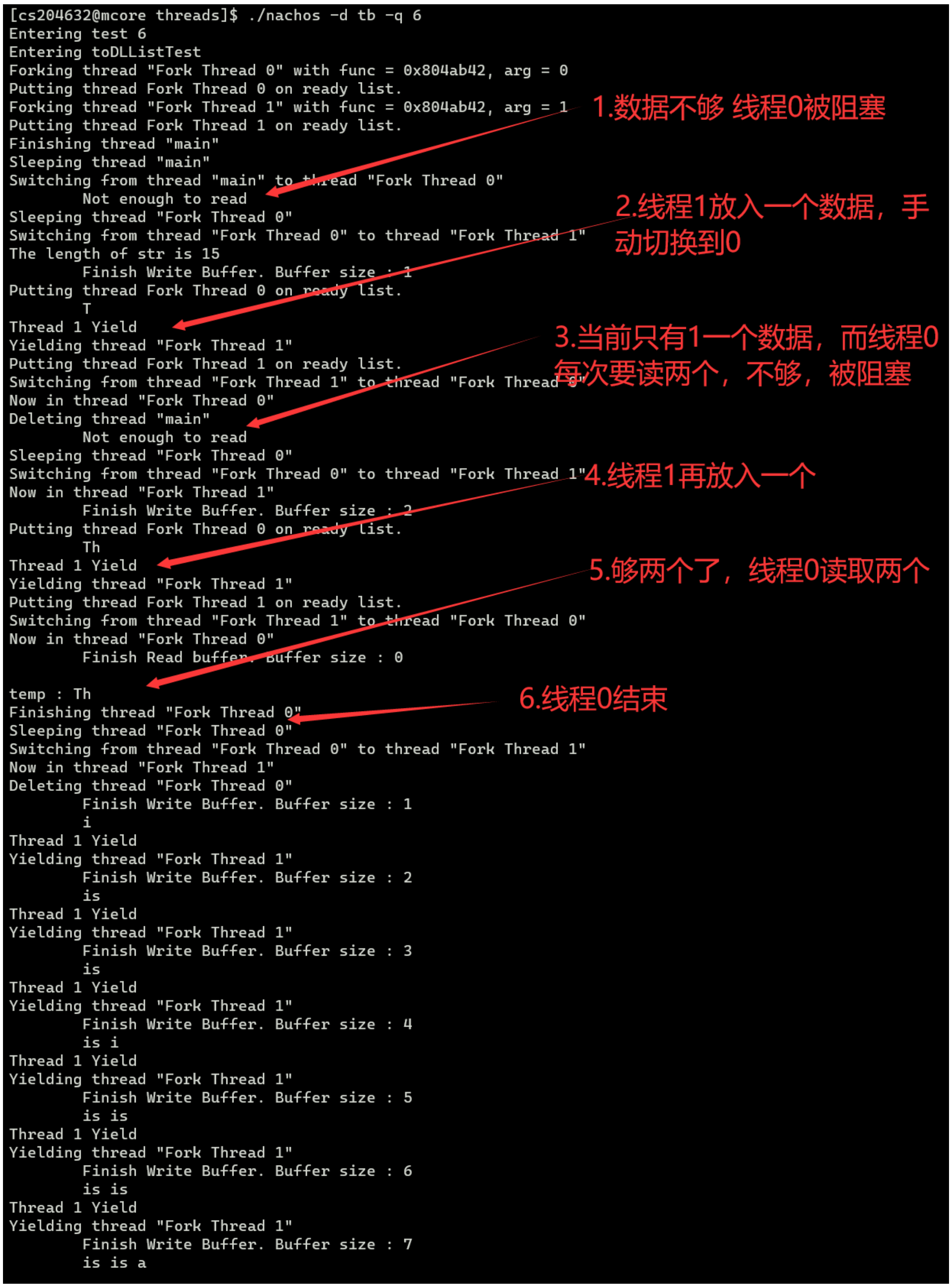
先读取，但buffer中数据量不够，Thread0被阻塞
Thread1向buffer中写入，写完后无任务，则Sleep
在写入第一次向buffer中写入数据的时候就可以看到Thread0被放入就绪队列中
Thread1结束后，Thread0从buffer中读取

```
[cs204632@mcore threads]$ ./nachos -d tb -q 6
Entering test 6
Entering toDLListTest
Forking thread "Fork Thread 0" with func = 0x804ab42, arg = 0
Putting thread Fork Thread 0 on ready list.
Forking thread "Fork Thread 1" with func = 0x804ab42, arg = 1
Putting thread Fork Thread 1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "Fork Thread 0"
        Not enough to read
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
The length of str is 15
        Finish Write Buffer. Buffer size : 1
Putting thread Fork Thread 0 on ready list.
        T
        Finish Write Buffer. Buffer size : 2
        Th
        Finish Write Buffer. Buffer size : 3
        Thi
        Finish Write Buffer. Buffer size : 4
        This
        Finish Write Buffer. Buffer size : 5
        This
        Finish Write Buffer. Buffer size : 6
        This i
        Finish Write Buffer. Buffer size : 7
        This is
        Finish Write Buffer. Buffer size : 8
        This is
        Finish Write Buffer. Buffer size : 9
        This is a
        Finish Write Buffer. Buffer size : 10
        This is a
        Finish Write Buffer. Buffer size : 11
        This is a t
        Finish Write Buffer. Buffer size : 12
        This is a te
        Finish Write Buffer. Buffer size : 13
        This is a tes
        Finish Write Buffer. Buffer size : 14
        This is a test
        Finish Write Buffer. Buffer size : 15
        This is a test

Finishing thread "Fork Thread 1"
Sleeping thread "Fork Thread 1"
Switching from thread "Fork Thread 1" to thread "Fork Thread 0"
Now in thread "Fork Thread 0"
Deleting thread "Fork Thread 1"
        Finish Read buffer. Buffer size : 13
        is is a test

temp : Th
        Finish Read buffer. Buffer size : 11
         is a test

temp : is
        Finish Read buffer. Buffer size : 9
        s a test

temp :  i
        Finish Read buffer. Buffer size : 7
        a test

temp : s
        Finish Read buffer. Buffer size : 5
        test

temp : a
        Finish Read buffer. Buffer size : 3
        st

temp : te
        Finish Read buffer. Buffer size : 1


temp : st
```

- 手动切换线程

加了一行 currentThread->Yield()，Thread1 会放弃 CPU，Thread0 会接管。

```
void BoundedBufferTest(int which) {
    if(which % 2) {
        char *str = "This is a test\n";
        printf("The length of str is %d\n", strlen(str));
        for(int i = 0; i < 100; i++) {
            char temp = str[i];
            bb->Write(&temp, 1);
            bb->Print();
            DEBUG('b', "Thread %d Yield\n", which);
            currentThread->Yield();
            if(temp == '\n') {
                break;
            }
        }
    } else {
        for(int i = 0; i < 100; i++) {
            char *temp = new char[3];
            temp[2] = '\0';
            bb->Read(temp, 2);
            bb->Print();
            printf("temp : %s\n", temp);
            if(bb->getUsed() == 0) break;
        }
    }
}
```

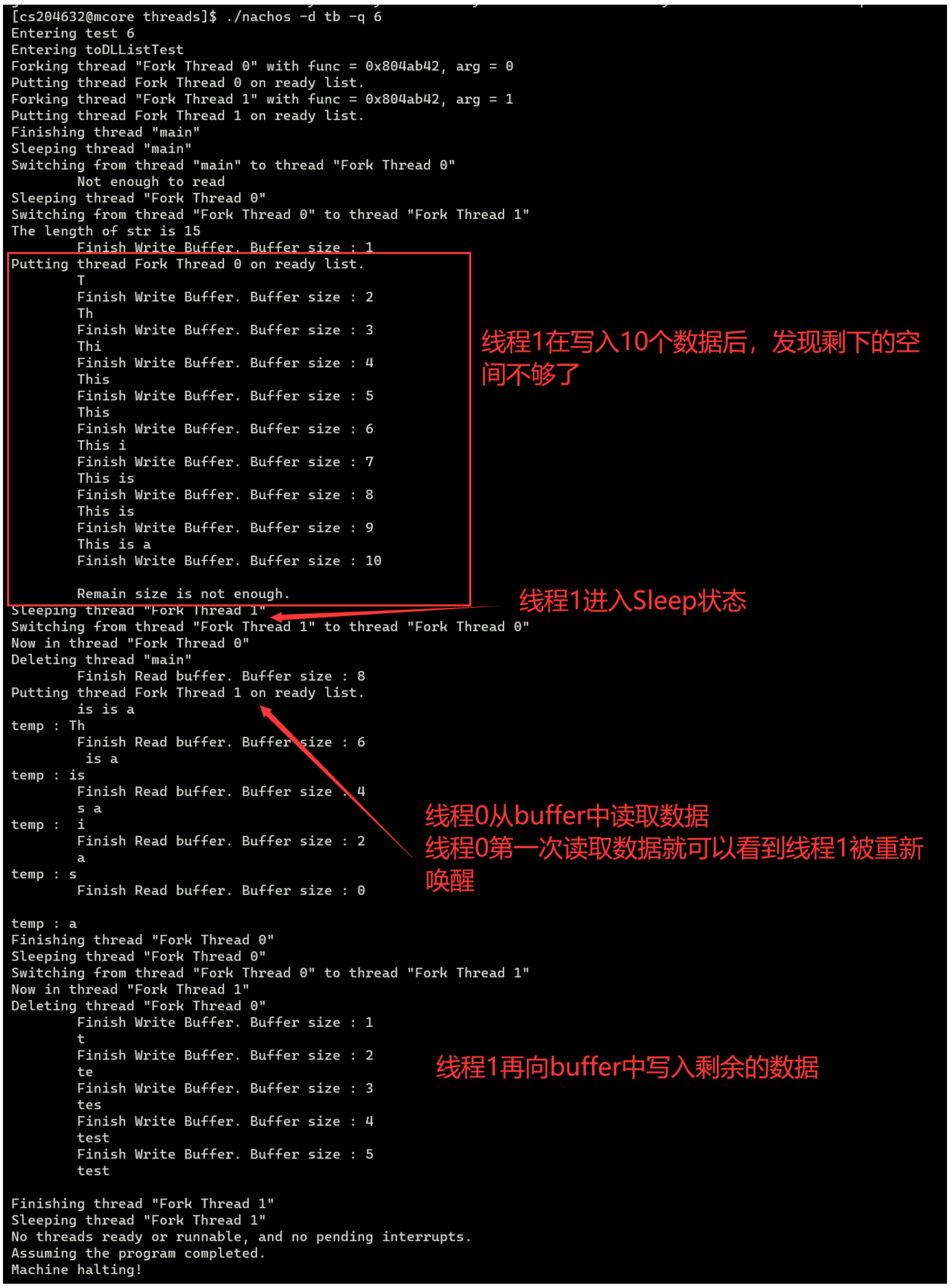


```
[cs204632@mcore threads]$ ./nachos -d tb -q 6
Entering test 6
Entering toDLListTest
Forking thread "Fork Thread 0" with func = 0x804ab42, arg = 0
Putting thread Fork Thread 0 on ready list.
Forking thread "Fork Thread 1" with func = 0x804ab42, arg = 1
Putting thread Fork Thread 1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "Fork Thread 0"
        Not enough to read
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
The length of str is 15
        Finish Write Buffer. Buffer size : 1
Putting thread Fork Thread 0 on ready list.
        T
Thread 1 Yield
Yielding thread "Fork Thread 1"
Putting thread Fork Thread 1 on ready list.
Switching from thread "Fork Thread 1" to thread "Fork Thread 0"
Now in thread "Fork Thread 0"
Deleting thread "main"
        Not enough to read
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
Now in thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 2
Putting thread Fork Thread 0 on ready list.
        Th
Thread 1 Yield
Yielding thread "Fork Thread 1"
Putting thread Fork Thread 1 on ready list.
Switching from thread "Fork Thread 1" to thread "Fork Thread 0"
Now in thread "Fork Thread 0"
        Finish Read buffer. Buffer size : 0

temp : Th
Finishing thread "Fork Thread 0"
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
Now in thread "Fork Thread 1"
Deleting thread "Fork Thread 0"
        Finish Write Buffer. Buffer size : 1
        i
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 2
        is
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 3
        is
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 4
        is i
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 5
        is is
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 6
        is is
Thread 1 Yield
Yielding thread "Fork Thread 1"
        Finish Write Buffer. Buffer size : 7
        is is a
```

1.数据不够 线程0被阻塞
2.线程1放入一个数据，手动切换到0
3.当前只有1一个数据，而线程0每次要读两个，不够，被阻塞
4.线程1再放入一个
5.够两个了，线程0读取两个
6.线程0结束

- buffer full

测试这一部分，我只将 buffer 的 maxSize 改为了 10。



```
[cs204632@mcore threads]$ ./nachos -d tb -q 6
Entering test 6
Entering toDLListTest
Forking thread "Fork Thread 0" with func = 0x804ab42, arg = 0
Putting thread Fork Thread 0 on ready list.
Forking thread "Fork Thread 1" with func = 0x804ab42, arg = 1
Putting thread Fork Thread 1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "Fork Thread 0"
        Not enough to read
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
The length of str is 15
        Finish Write Buffer. Buffer size : 1
Putting thread Fork Thread 0 on ready list.
        T
        Finish Write Buffer. Buffer size : 2
        Th
        Finish Write Buffer. Buffer size : 3
        Thi
        Finish Write Buffer. Buffer size : 4
        This
        Finish Write Buffer. Buffer size : 5
        This
        Finish Write Buffer. Buffer size : 6
        This i
        Finish Write Buffer. Buffer size : 7
        This is
        Finish Write Buffer. Buffer size : 8
        This is
        Finish Write Buffer. Buffer size : 9
        This is a
        Finish Write Buffer. Buffer size : 10

        Remain size is not enough.
Sleeping thread "Fork Thread 1"
Switching from thread "Fork Thread 1" to thread "Fork Thread 0"
Now in thread "Fork Thread 0"
Deleting thread "main"
        Finish Read buffer. Buffer size : 8
Putting thread Fork Thread 1 on ready list.
        is is a
temp : Th
        Finish Read buffer. Buffer size : 6
         is a
temp : is
        Finish Read buffer. Buffer size : 4
        s a
temp :  i
        Finish Read buffer. Buffer size : 2
        a
temp : s
        Finish Read buffer. Buffer size : 0

temp : a
Finishing thread "Fork Thread 0"
Sleeping thread "Fork Thread 0"
Switching from thread "Fork Thread 0" to thread "Fork Thread 1"
Now in thread "Fork Thread 1"
Deleting thread "Fork Thread 0"
        Finish Write Buffer. Buffer size : 1
        t
        Finish Write Buffer. Buffer size : 2
        te
        Finish Write Buffer. Buffer size : 3
        tes
        Finish Write Buffer. Buffer size : 4
        test
        Finish Write Buffer. Buffer size : 5
        test

Finishing thread "Fork Thread 1"
Sleeping thread "Fork Thread 1"
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

线程1在写入10个数据后，发现剩下的空间不够了

线程1进入Sleep状态

线程0从buffer中读取数据
线程0第一次读取数据就可以看到线程1被重新唤醒

线程1再向buffer中写入剩余的数据

四、遇到的问题

- 在构造线性表时，一开始未定义索引 index 而是想直接删除表中最后一个元素，使用了 tableBase[cnt-1]。但是 cnt 只是记录表中元素个数的变量，由于

表中的元素可能不连续，所以有可能出现阻塞，直到恰好该位置被添加元素。但是一旦该位置一直为空，就有可能出现饥饿。

- 在做最后一个部分实现线程安全的缓冲区时，用一般的方法不太能很好看出线程的切换过程，于是我使用了 DEBUG 的方法来看线程是如何切换的，可以看到切换的流程可以很清晰的看出来。
- 考虑多个线程同时访问同一个 BoundedBuffer 的情况，可能出现线程 A 写入，线程 B 和线程 C 都去读取，我们只能让一个线程去读取。所以在实现缓冲区的时候，不仅仅有同步的问题，互斥的问题也同样存在。

五、实验总结

- 在本次操作系统实验中，我学习了两种实现锁机制和条件变量的方式：Sleep 加中断和信号量。在使用 Sleep 加中断的方式时，我使用了一个标志位来表示锁的状态，当锁被占用时，其他线程需要等待直到锁被释放。在释放锁的时候，我使用了中断的方式，以保证操作的原子性。
- 另一种方式是使用信号量。信号量是一种计数器，用于控制多个线程对共享资源的访问。在本次实验中，我使用了信号量来实现锁机制和条件变量。当信号量的值大于 0 时，表示锁是可用的；当信号量的值为 0 时，表示锁被占用。
- 在实际的事件中，实现锁的方式的多样的，可以利用信号量去实现一个锁的机制，也可以简单的使用一个状态去记录所得状态。
- 在这个过程之中，都需要保证操作的原子性，也就是说在改变锁的状态时，不能被打断，所以需要使用关闭中断的方式去保证操作的原子性。这样做也有一定的弊端，关闭中断会使得系统变成一个无法被抢占的状态，可以在某些情况下导致死锁。
- 除了实现锁机制和条件变量，我还将双向链表改成了线程安全的形式，并且实现了一个线程安全的表结构。在这个过程中，我使用了锁机制和条件变量来保证多个线程同时访问数据结构时不会出现数据竞争和其他问题。最后，

我还实现了一个大小受限的缓冲区，通过使用锁机制和条件变量来保证多个线程同时访问缓冲区时不会出现问题。

- 总的来说，本次操作系统实验让我更加深入地了解了多线程编程和操作系统底层原理。通过实践，我掌握了多种实现锁机制和条件变量的方式，并且学会了如何将数据结构改造成线程安全的形式。这些经验将对我的未来学习和工作产生积极的影响。