实验二 线程与同步

吴朱冠宇 22920182204320　胡子潇 22920182204188
谭心怡 22920182204283　卞珂 22920182204133

1 实验简述

本次实验的目的在于将 nachos 中的锁机制和条件变量的实现补充完整，并利用这些同步机制实现几个基础工具类。实验内容分三部分：实现锁机制和条件变量，并利用这些同步机制将实验一中所实现双向有序链表类修改成线程安全的；实现一个线程安全的表结构；实现一个大小受限的缓冲区。

2 实验内容

2.1 用 `Thread::Sleep` 实现锁机制和条件变量

1. 问题描述

锁和条件变量是实现线程同步与互斥的重要机制，锁可以保证临界区安全，条件变量用于阻塞线程直到特定条件为真。这部分实现主要参考 `Semaphore` 的实现，在必要的时候应关中断，阻塞进程应该挂到相应的阻塞队列中，不同的等待事件应对应不同的阻塞队列。

2. 具体实现

(1) 用 `Thread::Sleep` 实现锁机制

1. `Lock` **类定义与初始化 思路描述：**

根据教材，实现互斥必须满足以下要求：

- 必须强制实施互斥：在与相同资源或共享对象的临界区有关的所有进程中，一次只允许一个进程进入临界区。
- 一个在非临界区停止的进程不能干涉其他进程。
- 不允许出现需要访问临界区的进程被无限延迟的情况，即不会死锁或饥饿。
- 没有进程在临界区中时，任何需要进入临界区的进程必须能够立即进入。
- 对相关进程的执行速度和处理器的数量没有任何要求和限制。
- 一个进程驻留在临界区中的时间必须是有限的。

对于要求 1、2、4，可以设置一个双值型变量 `mutex`，通过其不同的取值表示锁的开关状态（分别对应临界区内有无线程），这里用 `0` 和 `1` 分别表示开和关。

对于要求 5、6，由于开关中断是通过函数 `Lock::Acquire ()` 和 `Lock::Release ()` 实现而非用户自由控制，因此只需使得以上两个函数（分别实现获得锁、释放锁）中间过

程正确，线程就会被中断。

对于要求 3，只需要在满足上述要求的基础上，添加一个等待队列 `queue` 和一个方法 `Thread::Sleep` 即可。一个线程无法进入临界区时，将其放入等待线程队列，令其休眠并将其执行权交由其他线程。基于队列先进先出的特性，进程必然会按照顺序向后执行，因此不会出现进程被无限延迟的情况。

参考所给出的 nachos 英文文档 3.2.5 中给出的提示，另有以下两个要求：

- What will happen if a lock holder attempts to acquire a held lock a second time? What if a thread tries to release a lock that it does not hold?
- 如果一个锁持有者再次尝试获取一个被持有的锁，会发生什么情况？如果一个线程试图释放它没有持有的锁，该怎么办？

即已获得锁的线程不可再次要求获得锁；未获得锁的线程不能要求释放锁。因此我们设置了 `heldThread` 属性来记录当前获得锁的线程指针，若没有线程获得锁则为 NULL。此外，还有一个用于调试的属性 `name` 来标记锁。

根据以上对于所得定义，可以实现 `Lock` 类的构造和析构函数：

```
//Lock类定义
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();   // true if the current thread
                    // holds this lock.  Useful for
                    // checking in Release, and in
                    // Condition variable ops below.

  private:
    char* name;             // for debugging
    // plus some other stuff you'll need to define
    int mutex;              // 0表示开，1表示关
    List *queue;
    Thread *heldThread;     // 当前获得锁的线程指针
                            // 若没有线程获得锁则为NULL
};
```

```
//Lock类构造与析构函数
Lock::Lock(char* debugName) {
    name = debugName;   //根据传入参数设置name属性
```

```
    mutex = 0;          //将mutex置为0（打开）
    queue = new List;   //给queue分配内存
    heldThread = NULL;  //将heldThread置为NULL（没有线程拥有锁）
}
Lock::~Lock() {
    delete queue;       //释放队列的内存，防止内存泄漏
}
```

2. `Lock::Acquire()` **函数实现**

思路描述：

根据对 `Lock` 类的解释，我们需要在 `Lock::Acquire` 函数的始末打开和关闭系统中断，对 `mutex` 判断和操作的一致性。

另，根据 Nachos Project Guide 中 3.2.5 节的提示，不能重复要求获得锁：我们在 `Lock::Acquire` 函数的最开始处，通过 `ASSERT(!isHeldByCurrentThread())` 来确保锁被正确的使用。

当 `mutex` 不为 0 时（也就是锁被占有），将当前线程加入等待队列，令其休眠，以此来保证其他线程可以切换到执行态且当前线程可以在稍后被正确唤醒。当 `mutex` 为 0 时，将 `mutex` 置为 1（占有锁），并将 `heldThread` 置为 `currentThread`（系统提供的用于标识当前线程的指针）。

```
void Lock::Acquire() {
    ASSERT(!isHeldByCurrentThread()); // 当前线程并未占有锁
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // 禁用中断
    while (mutex) { // 锁被占有
        queue->Append((void *)currentThread);
        currentThread->Sleep();
    }
    mutex = 1; // 上锁
    heldThread = currentThread;             // 当前线程占有锁
    (void) interrupt->SetLevel(oldLevel);   // 重新开中断
}
```

3. `Lock::Release()` **函数实现**

思路描述：

同理，对于 `Lock::Release` 函数我们也通过开关中断来保证中间过程的正确性。

并仍然通过 `ASSERT(!isHeldByCurrentThread())` 来确保满足锁被正确使用的另一条件——不能释放非本线程拥有的锁。

在释放锁的过程中需要通过从等待队列取出一个当前正在等待的线程，若当前队列中存在线程则将其取出并唤醒，否则不需要唤醒线程；然后需要将 `mutex` 置为 0（释放锁），`heldThread` 置为 NULL，表示没有线程占有锁。

```
void Lock::Release() {
    ASSERT(isHeldByCurrentThread()); // 当前线程占有锁
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // 禁用中断
    Thread *thread = (Thread *)queue->Remove(); // 取sleep队列头部的线程
    if (thread != NULL) {
        scheduler->ReadyToRun(thread);  // 唤醒线程，进入准备运行态
    }
    mutex = 0; // 释放锁
    heldThread = NULL; // 无线程占有锁
    (void) interrupt->SetLevel(oldLevel);   // 重新开中断
}
```

4. **Lock::isHeldByCurrentThread() 函数实现**

思路描述：

为保证获得和释放锁的过程中对 heldThread 的正确修改，将 heldThead 与当前线程指针 currentThread 进行比较，返回拥有锁的线程是否为当前线程的判断结果。

```
bool Lock::isHeldByCurrentThread() {
    return currentThread == heldThread;
}
```

（2）用 Thread::Sleep 实现条件变量

1. **Condition 类定义与初始化**

思路描述：

条件变量需在条件不成立时释放锁，并令当前线程休眠。因此，为保证条件变量的正确使用，Condition::Wait、Condition::Signal、Condition::Broadcast 所使用的锁必须是同一个锁，我们设置了 heldLock 来记录当前条件变量所使用的锁。此外，还有一个用于调试的 name 变量，用来标识条件变量。

同时，在实现构造函数时，我们首先根据传入参数初始化 name，并将 heldLock 置为 NULL（不拥有任何锁），并给 queue 分配内存；在析构函数 Condition::~Condition() 中释放 queue 的内存空间。

```
class Condition {
  public:
    Condition(char* debugName);     // initialize condition to
                    // "no one waiting"
    ~Condition();           // deallocate the condition
    char* getName() { return (name); }

    void Wait(Lock *conditionLock);     // these are the 3 operations on
                    // condition variables; releasing the
```

```
                    // lock and going to sleep are
                    // *atomic* in Wait()
    void Signal(Lock *conditionLock);   // conditionLock must be held by
    void Broadcast(Lock *conditionLock);// the currentThread for all of
                    // these operations

  private:
    char* name;
    // plus some other stuff you'll need to define
    List *queue;    // 等待的线程队列
    Lock *heldLock; // 记录Condition::wait()函数中的锁
};
```

```
Condition::Condition(char* debugName) {
    name = debugName;
    queue = new List;
    heldLock = NULL;
}
Condition::~Condition() {
    delete queue;
}
```

2. **Condition::Wait() 函数实现**

思路描述:

首先释放锁，将当前线程加入等待队列，然后令进程休眠（因为 Nachos 要求再休眠前必须先关闭中断，所以我们再休眠前关闭中断），当进程被唤醒时需要再重新获得锁。

最后，为了保证 Condition::Signal 和 Condition::Broadcast 使用的锁与 Condition::Wait 一致，我们根据锁的状态修改 heldLock 的值，用于后面两个函数中进行判断。

```
void Condition::Wait(Lock* conditionLock) {
    conditionLock->Release(); // 释放当前占据的锁，准备进入休眠
    heldLock = NULL;
    queue->Append((void *)currentThread); // 当前线程加入休眠队列
    // nachos要求线程休眠前必须要关中断
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    currentThread->Sleep();
/*---------- 进入readtorun 后重新被唤醒 ----------*/
    conditionLock->Acquire(); // 重新获得锁
    heldLock = conditionLock;
    (void) interrupt->SetLevel(oldLevel);   // 重新开中断
}
```

3. **Condition::Signal() 函数实现**

思路描述：

首先，使用 ASSERT 来保证条件变量被正确使用。接着，我们从等待队列中取出一个休眠的线程，若队列非空，则运行取出的线程。

```
void Condition::Signal(Lock* conditionLock) {
    ASSERT(heldLock == conditionLock || heldLock == NULL);
    Thread *thread = (Thread *)queue->Remove();
    if (thread != NULL) {
        scheduler->ReadyToRun(thread);
    }
}
```

4. `Condition::Broadcast()` **函数实现**

思路描述：

与 Condition::Signal 类似，唯一不同的是唤醒等待队列中的所有线程。

```
void Condition::Broadcast(Lock* conditionLock) {
    ASSERT(heldLock == conditionLock);
    Thread *thread = (Thread *)queue->Remove();
    while (thread != NULL) {
        scheduler->ReadyToRun(thread);
        thread = (Thread *)queue->Remove();
    }
}
```

2.2 用 Samaphore 实现锁机制和条件变量

1. 问题描述

信号量（Semaphore）和锁配合条件变量是实现线程同步与互斥的两种等价方式。在本实验中，我们尝试用 nachos 系统的信号量实现锁机制和条件变量。（**注意 Semaphore 与条件变量的区别：** 如果在调用 `Semaphore::P()` 前调用 `Semaphore::V()`，则 V 操作的效果将积累下来；而如果在调用 `Condition::Wait()` 前调用 `Condition::Signal()`，则 Signal 操作的效果将不积累。）

2. 具体实现

（1）用 Semaphore 实现锁机制

1. **Lock 类定义与初始化**

思路描述：

用一个值为 1 的信号量实现互斥锁，因此类定义中定义一个信号量并初始化为 1。另外，由于互斥锁要求为其加锁的进程和为其解锁的线程必须为同一个线程，因此定义 heldThread 来记录持有锁的进程，并将其初始化为 NULL，表示没有线程持有该锁。

```
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();   // true if the current thread
                    // holds this lock.  Useful for
                    // checking in Release, and in
                    // Condition variable ops below.

  private:
    char* name;             // for debugging
    // plus some other stuff you'll need to define
    Semaphore *sem;       // 用一个值为1的信号量表示互斥锁
    Thread *heldThread;   // 当前获得锁的线程指针
                          // 若没有线程获得锁则为NULL
};
```

```
Lock::Lock(char* debugName) {
    name = debugName;
    heldThread = NULL;
    sem = new Semaphore(name, 1); // 初始资源数为1
}
Lock::~Lock() {
    delete sem;
}
```

2. `Lock::Acquire()` 函数实现

思路描述：

对信号量执行 P 操作，信号量值减 1 变为 0，并将当前线程设置为该锁的拥有者。当另一个线程请求锁时，再次执行 P 操作，由于此时信号量值为 0，线程被阻塞。这样就保证了只有一个线程拥有锁。

```
void Lock::Acquire() {
    ASSERT(!isHeldByCurrentThread());
    sem->P();
    heldThread = currentThread;
}
```

3. `Lock::Release()` 函数实现

思路描述：

当锁的所有线程释放锁时，对信号量执行 V 操作，唤醒一个阻塞线程，并将信号量的值加 1 变回 1，heldThread 重新置为 NULL，这样就回到初始状态，等待下一次再被请求。

```
void Lock::Release() {
    ASSERT(isHeldByCurrentThread());
    sem->V();
    heldThread = NULL;
}
```

4. `Lock::isHeldByCurrentThread()` 函数实现

思路描述：

只需比较当前获得锁的线程与当前线程是否是相同即可。

```
bool Lock::isHeldByCurrentThread() {
    return currentThread == heldThread;
}
```

（2）用 Semaphore 实现条件变量

1. **Condition 类定义与初始化**

思路描述：

环境变量可以用一个值为 0 的信号量实现，同时定义 blockCnt 记录阻塞线程数。另外，定义 heldLock 以辅助检测环境变量与互斥锁是否相关联。

```
class Condition {
  public:
    Condition(char* debugName);     // initialize condition to
                    // "no one waiting"
    ~Condition();           // deallocate the condition
    char* getName() { return (name); }

    void Wait(Lock *conditionLock);     // these are the 3 operations on
                    // condition variables; releasing the
                    // lock and going to sleep are
                    // *atomic* in Wait()
    void Signal(Lock *conditionLock);   // conditionLock must be held by
    void Broadcast(Lock *conditionLock);// the currentThread for all of
                    // these operations
```

```
  private:
    char* name;
    // plus some other stuff you'll need to define
    int blockCnt;
    Semaphore *sem;
    Lock *heldLock;
};
```

```
Condition::Condition(char* debugName) {
    name = debugName;
    blockCnt = 0;
    heldLock = NULL;
    sem = new Semaphore(debugName, 0);
}
Condition::~Condition() {
    delete sem;
}
```

2. Condition::Wait() 函数实现

思路描述：

首先记录下当前的互斥锁，然后解开相应的互斥锁并执行 P 操作，等待条件发生变化。一旦其它的某个线程改变了条件变量，唤醒一个阻塞线程并重新上锁。

```
void Condition::Wait(Lock* conditionLock) {
    heldLock = conditionLock;
    conditionLock->Release();
    blockCnt++;
    sem->P();
    conditionLock->Acquire();
}
```

3. Condition::Signal() 函数实现

思路描述：

在验证当前互斥锁与环境变量相关联之后，如果存在阻塞进程，则执行 V 操作通知相应的条件变量唤醒线程。

```
void Condition::Signal(Lock* conditionLock) {
    ASSERT(heldLock == conditionLock || heldLock == NULL);
    if (blockCnt) {
        blockCnt--;
        sem->V();
```

```
    }
}
```

4. Condition::Broadcast() 函数实现

思路描述：

与 Condition::Signal() 函数类似，只是通过循环唤醒所有被 wait 函数阻塞在某个条件变量上的线程。

```
void Condition::Broadcast(Lock* conditionLock) {
    ASSERT(heldLock == conditionLock || heldLock == NULL);
    while (blockCnt) {
        blockCnt--;
        sem->V();
    }
}
```

2.3 用锁机制和条件变量修改双向有序链表

1. 问题描述

前一个实验在 nachos 系统中运行了自己编写的双向链表程序，并且演示了一些并发错误，出现这些错误的原因是测试程序未考虑互斥。现在根据所实现的锁和条件变量机制实现线程安全的双向链表 SynchDLList，确保其多线程并发程序是正确互斥的。

2. 具体实现

（1） **SynchDLList 类定义与初始化**

思路描述：

SynchDLList 实际上是给 DLList 套一层外壳，其核心是一个 `DLList *list`。实现线程安全需额外定义一个锁和一个用于判断链表非空的条件变量。

```
class SynchDLList {
public:
    SynchDLList(int errType = 0);    // 构造函数
    ~SynchDLList();  // 析构函数
    void *Remove(int *keyPtr);  // remove from head of list
                                // set *keyPtr to key of the removed item
                                // return item (or NULL if list is empty)
    void SortedInsert(void *item, int sortKey);
    void printList();

private:
```

```
    DLList *list;
    Lock *lock;
    Condition *empty;
};
```

```
SynchDLList::SynchDLList(int errType) {
    list = new DLList(errType);
    lock = new Lock("Lock for DLList");
    empty = new Condition("List empty");
}

SynchDLList::~SynchDLList() {
    delete list;
    delete lock;
    delete empty;
}
```

(2) SynchDLList::Remove() 函数实现

思路描述：

实现线程安全的链表头部结点删除，一是保证链表互斥访问，为此需要加锁；二是不能操作空链表，为此需要通过条件变量阻塞等待链表非空。

```
void *SynchDLList::Remove(int *keyPtr) {
    void *item;
    lock->Acquire();
    while (list->IsEmpty()) {
        empty->Wait(lock);
    }
    item = list->Remove(keyPtr);
    lock->Release();
    return item;
}
```

(3) SynchDLList::SortedInsert() 函数实现

思路描述：

实现线程安全的链表结点插入，只需要加锁保证链表互斥访问。另外，成功结点之后链表必定为空，此时可以唤醒一个被条件变量阻塞的线程。

```
void SynchDLList::SortedInsert(void *item, int sortKey) {
    lock->Acquire();
    list->SortedInsert(item, sortKey);
```

```
    empty->Signal(lock);
    lock->Release();
}
```

(4) SynchDLList::printList() 函数实现

思路描述:

逐个遍历打印即可。

```
void SynchDLList::printList() {
    list->printList();
}
```

(5) **修改** dllist-driver.cc

思路描述:

除了需要将出现的 DLList 类修改为 SynchDLList 类之外，dllist-driver.cc 中的 RemoveList() 函数不再需要判断链表非空。

```
void InsertList(int threadNum, int N, SynchDLList *slist) {
    RandomInit(unsigned(time(0) + threadNum));
    for (int i = 0; i < N; i++) {
        void *item;
        int key = Random() % 100;          //生成节点的随机值
        int *items = new int[1];
        items[0] = -key;
        item = items;
        printf("Thread %d: Inserted key %2d\n", threadNum, key);
        slist->SortedInsert(item, key);
        printf("Thread %d: ", threadNum);
        slist->printList();
    }
}

void RemoveList(int threadNum, int N, SynchDLList *slist) {
    for (int i = 0; i < N; i++) {
        int key;
        printf("Thread %d: Removed  key ", threadNum);
        slist->Remove(&key);
        printf("%2d\n", key);
        printf("Thread %d: ", threadNum);
        slist->printList();
    }
}
```

(6) **修改 threadtest.cc**

同样的，threadTest2() 函数中的链表也改成 SynchDLList 类。

```
void DLListThread(int t) {
    InsertList(t, insCnt, sdllist);
    RemoveList(t, insCnt, sdllist);
}
void ThreadTest2() {
    DEBUG('t', "Entering ThreadTest2");
    sdllist = new SynchDLList(errType);
    for (int i = 1; i < threadNum; i++) {
        Thread *t = new Thread("forker thread");
        t->Fork(DLListThread, i);
    }
    DLListThread(threadNum);
}
```

3. 实验结果

(1) **用 Thread::Sleep 实现锁机制和条件变量**

如图1和图2所示，分别使用./nachos -q 2 -n 2 -t 2 -e 1和./nachos -q 2 -n 2 -t 2 -e 2两条命令，进行2个线程各插入删除2个数的操作测试。可以看见，尽管存在线程切换，插入删除依然按照线程顺序完成，不存在混乱和错误。

```
[cs182204320@mcore threads]$ ./nachos -q 2 -n 2 -t 2 -e 1
Thread 2: Inserted key  8
SortedInsert error, Switching before inserting
Thread 1: Inserted key 89
Thread 2: Current List:  8

Thread 2: Inserted key 29
SortedInsert error, Switching before inserting
Thread 2: Current List:  8 29

Thread 2: Removed  key  8
Thread 2: Current List: 29

Thread 2: Removed  key 29
Thread 2: Current List:


SortedInsert error, Switching before inserting
Thread 1: Current List: 89

Thread 1: Inserted key 20
SortedInsert error, Switching before inserting
Thread 1: Current List: 20 89

Thread 1: Removed  key 20
Thread 1: Current List: 89

Thread 1: Removed  key 89
Thread 1: Current List:

No threads ready or runnable, and no pending interrupts.
```

图 1: 使用 Thread::Sleep 上锁测试 Error 1

```
[cs182204320@mcore threads]$ ./nachos -q 2 -n 2 -t 2 -e 2
Thread 2: Inserted key 19
SortedInsert error, insert one and switch immediately
Thread 1: Inserted key  4
Thread 2: Current List: 19

Thread 2: Inserted key 50
Thread 2: Current List: 19 50

Thread 2: Removed  key 19
Thread 2: Current List: 50

Thread 2: Removed  key 50
Thread 2: Current List:


SortedInsert error, insert one and switch immediately
Thread 1: Current List:  4

Thread 1: Inserted key  6
Thread 1: Current List:  4  6

Thread 1: Removed  key  4
Thread 1: Current List:  6

Thread 1: Removed  key  6
Thread 1: Current List:

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

图 2: 使用 Thread::Sleep 上锁测试 Error 2

(2) **用 Thread::Semaphore 实现锁机制和条件变量**

如图3和图4所示，运行上面的两条命令，同样不存在混乱和错误。

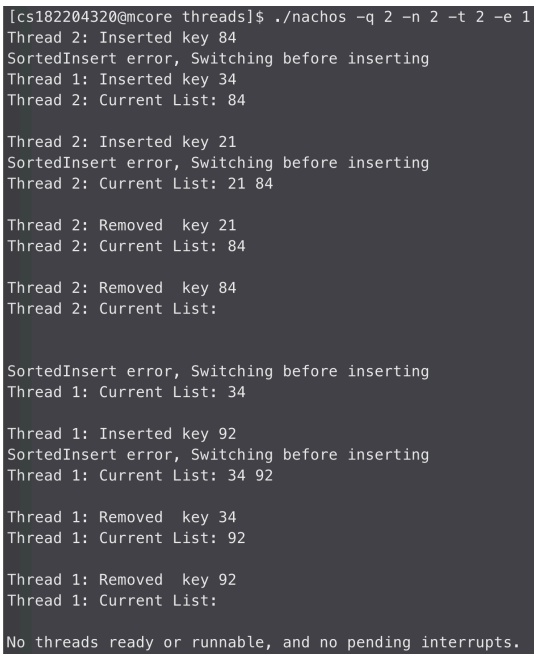

图 3: 使用 `Thread::Sem` 上锁测试 Error 1

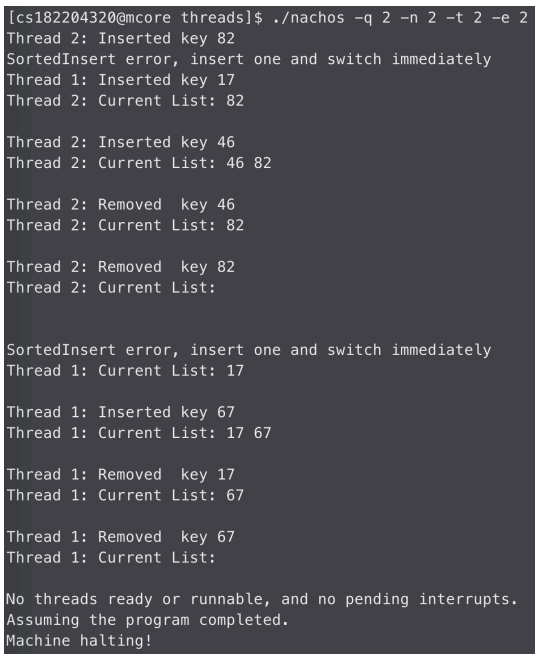

图 4: 使用 `Thread::Sem` 上锁测试 Error 2

2.4 实现一个线程安全的表结构

1. 实验要求与分析

该实验要求实现一个线程安全的表结构。要求在有多个线程对同一个表操作时，仍然能够保证每个线程对表的操作（Alloc，Get 和 Release）是“原子操作”，即不会因为线程的切换等原因而发生错误。

为了保证这三个操作在执行期间不被打断，我们需要使用锁机制来进行对临界区域代码的保护。而由于该实验中只需要使用信号量，因此对于上面实现的两种同步机制，我们的处理方式均是一样的。

2. 代码实现

(1) **对原 `Table.h` 进行更正修改**

思路描述：

更正原构造函数声明，添加构析函数声明，并添加表示表的最大可用槽数的变量 tableSize、互斥锁 tableLock、存储数据的表结构 table 和标识当前表槽是否已经被使用的 used：

```
class Table {
  public:
    Table(int size);
    ~Table();

    int Alloc(void *object);
```

```
    void *Get(int index);
    void Release(int index);

  private:
    // Your code here.
    int tableSize;
    Lock *tableLock;
    void **table;
    int *used;
};
```

（2） **构造函数** `Table::Table(int size)`

思路描述：

构造函数要做的事情应当包括设置最大表槽可用数 tableSize、创建一个互斥锁、创建一个存储数据的表结构、创建一个标识表槽是否已被使用的 used 并对其进行初始化：

```
Table::Table(int size) {
    tableSize = size;
    tableLock = new Lock("tableLock");
    table = new void *[size];
    used = new int[size];
    for (int i = 0; i < size; i++) {
        used[i] = 0;
    }
}
```

（3） **析构函数** `Table::~Table()`

析构函数应当释放之前构造函数所申请的所有空间：

```
Table::~Table() {
    delete[] table;
    delete tableLock;
    delete used;
}
```

（4） `Table::Alloc()`

思路描述：

`Table::Alloc()` 函数用于向表中存储一个数据。根据 `Table.h` 中的定义可知，该表可以用来存储 `void *`（空指针）型的数据，在后续的实验中，本表结构可以被用于线程表、进程表等用途。因此，我们可以将要存入的数据首先强制转换为 `void *` 类型，然后利用标识 used，找到尚未被使用的表槽，然后将该数据元素插入目标位置（若表已满，则会报错处理）。

以上找到空位置并插入这一系列操作是不希望被打断的，因此我们需要在这些操作前后加上互斥锁来进行临界区的保护。而无论我们是否成功的存入了 object，都需要释放锁，以免其他线程无法获取资源运行。

```
int Table::Alloc(void *object) {
    tableLock->Acquire();           //获取锁
    //输出当前正在临界区内的线程，并指明当前线程的动作是Alloc
    printf("Thread\t%p\talloc:\t\t", currentThread);
    for (int i = 0; i < tableSize; i++) {
        if (!used[i]) {             //寻找尚未被使用的表槽
            used[i] = 1;            //置标识used为1
            table[i] = object;      //将数据存入表中
            printf("table[%d]:%d\n", i, *(int *)object);    //输出插入的位置和
                数值
            tableLock->Release();   //临界区代码执行完毕，释放锁
            return i;               //返回插入的位置
        }
    }
    printf("alloc fail\n");         //没有找到可以插入元素的位置，报错处理
    tableLock->Release();           //同样需要释放锁
    return -1;
}
```

(5) Table::Get()

思路描述：

此函数用来获取指定位置存储的数据的值，但不需要将该数值删去。在这一过程中，同样需要加锁来进行临界区的保护：

```
void *Table::Get(int index) {
    tableLock->Acquire();   //获取锁
    printf("Thread\t%p\tget:\t\t", currentThread);
    if (index >= 0 && index < tableSize) {
        if (used[index]) {
            printf("table[%d]:%d\n", index, *(int *)table[index]);  //输出相关
                信息
            tableLock->Release();   //释放锁
            return table[index];
        }
        else {
            printf("table[%d] is empty\n", index);  //该位置没有存储数值
        }
    }
    else {
        printf("index %d is out of range\n", index);    //要获取的位置不在表中
```

```
    }
    tableLock->Release();    //释放锁
    return NULL;
}
```

(6) `Table::Release()`

思路描述：

此函数用于将表中指定位置的数值删除，使该位置可以继续存储其他的数值，同样需要添加锁来进行临界区保护：

```
void Table::Release(int index) {
    tableLock->Acquire();       //获取锁
    printf("Thread\t%p\trelease:\t", currentThread);
    if (index >= 0 && index < tableSize) {
        if (used[index]) {  //该位置目前有数据，将其释放掉
            used[index] = 0;
            table[index] = NULL;
            printf("table[%d] is released\n", index);
        }
        else {
            printf("table[%d] is empty\n", index);
        }
    }
    else {
        printf("index %d is out of range\n", index);
    }
    tableLock->Release();    //释放锁
    return;
}
```

3. 测试

为了测试在多个线程随意切换的情况下，各个操作是否还可以保持其“原子性”，我们编写测试函数进行测试。测试函数中 fork 出若干线程，每个线程要做的事情就是首先向表中插入一个随机数字，然后尝试获取该数字，之后将该数值从表中删除。

如果我们的锁机制使用正确，那么存储、去读和删除均不会发生错误，每个线程可以正确的对自己生成的元素进行操作，而不会像上一次实验中的链表一样错误的处理了别的线程插入的元素。而且，若由于线程切换使得表槽被全部使用完毕，那么之后尝试插入时应当会有报错处理。

3.1 测试函数：

（1） 每个线程的动作：

```
void TableThread(int t) {
    int *num = new int[1];
    num[0] = Random() % 1000;    //获取随机数值
    void *ele = num;             //强制类型转换为void*
    int position = table->Alloc(ele);    //数据存入表中
    if (position != -1) {                //若插入失败了那么直接退出
        num[0] = *(int *)table->Get(position);  //获取自己存入的数据
        table->Release(position);               //删除自己插入的数据
    }
}
```

（2） 测试总函数：

```
void ThreadTest3() {
    DEBUG('t', "Entering ThreadTest3");
    RandomInit(unsigned(time(0)));
    table = new Table(itemNum);      //生成一个所有线程公用的表
    for (int i = 1; i < threadNum; i++) {        //创建若干个线程
        Thread *t = new Thread("forker thread");
        t->Fork(TableThread, i);
    }
    TableThread(threadNum);
}
```

3.2 测试结果：

（1） 输入：

`./nachos -rs -q 3 -t 线程数 -n 表槽数最大值`

注：`-rs` 表示多个线程可以进行随机切换

（1） 用 `Thread::Sleep` 实现锁机制和条件变量的情况下进行测试

测试命令及结果如图5所示。上述测试定义了 4 个并发线程，Table 最多可以存放 2 个数据，可以看到，虽然不同的线程之间有所切换，但是各个原子操作均没有受到影响。每个线程都是对自己存入的元素进行读取和删除，也并未发生“错位”的情况。

以上测试并未出现 Table 已满但还要插入的情况，下面测试这种情况：

测试命令及结果如图6所示。该测试中共并发了十个线程，而 Table 可用表槽数只有一个，同时由于线程切换，产生了 Table 已经满了但是还需要插入的情况，可以看到，在这种情况下，可以正常的产生插入失败的处理，而其他已经插入数据的线程和之后的线程不受影响可以正确的运行。

（1） 用 `Thread::Semaphore` 实现锁机制和条件变量的情况下进行测试

```
[cs182204283@mcore threads]$ ./nachos -rs -q 3 -t 10 -n 2
Thread  0x94260b8       alloc:          table[0]:32
Thread  0x94260b8       get:            table[0]:32
Thread  0x94260b8       release:        table[0] is released

Thread  0x9426180       alloc:          table[0]:123
Thread  0x942c1f8       alloc:          table[1]:991
Thread  0x942c1f8       get:            table[1]:991
Thread  0x942c1f8       release:        table[1] is released
Thread  0x9432270       alloc:          table[1]:715
Thread  0x9432270       get:            table[1]:715
Thread  0x9432270       release:        table[1] is released
Thread  0x94504c8       alloc:          table[1]:59
Thread  0x94504c8       get:            table[1]:59
Thread  0x94504c8       release:        table[1] is released
Thread  0x9456540       alloc:          table[1]:445
Thread  0x9456540       get:            table[1]:445
Thread  0x9456540       release:        table[1] is released
Thread  0x9426180       get:            table[0]:123
Thread  0x9426180       release:        table[0] is released
Thread  0x94382e8       alloc:          table[0]:535
Thread  0x94382e8       get:            table[0]:535
Thread  0x94382e8       release:        table[0] is released
Thread  0x943e360       alloc:          table[0]:742
Thread  0x943e360       get:            table[0]:742
Thread  0x943e360       release:        table[0] is released
Thread  0x94443d8       alloc:          table[0]:685
Thread  0x94443d8       get:            table[0]:685
Thread  0x94443d8       release:        table[0] is released
Thread  0x944a450       alloc:          table[0]:141
Thread  0x944a450       get:            table[0]:141
Thread  0x944a450       release:        table[0] is released
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

图 5: 线程数较少时 Table 的测试结果（Sleep 实现）

```
[cs182204283@mcore threads]$ ./nachos -rs -q 3 -t 10 -n 1
Thread  0x94980b8       alloc:          table[0]:833
Thread  0x94980b8       get:            table[0]:833
Thread  0x94980b8       release:        table[0] is released

Thread  0x9498180       alloc:          table[0]:615
Thread  0x949e1f8       alloc:          alloc fail
Thread  0x94aa2e8       alloc:          alloc fail
Thread  0x94b0360       alloc:          alloc fail
Thread  0x94b63d8       alloc:          alloc fail
Thread  0x94bc450       alloc:          alloc fail
Thread  0x94c24c8       alloc:          alloc fail
Thread  0x94c8540       alloc:          alloc fail
Thread  0x9498180       get:            table[0]:615
Thread  0x9498180       release:        table[0] is released
Thread  0x94a4270       alloc:          table[0]:308
Thread  0x94a4270       get:            table[0]:308
Thread  0x94a4270       release:        table[0] is released
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

图 6: 线程数较多时 Table 的测试结果（Sleep 实现）

将 synch.h 和 synch.cc 中的内容改为 synch-sem.cc 和 synch-sem.h 中的内容，重新编译，然后进行重复测试：

```
[cs182204283@mcore threads]$ ./nachos -rs -q 3 -t 10 -n 2
Thread  0x87780b8       alloc:          table[0]:602
Thread  0x87780b8       get:            table[0]:602
Thread  0x87780b8       release:        table[0] is released

Thread  0x8778178       alloc:          table[0]:53
Thread  0x877e1f0       alloc:          table[1]:917
Thread  0x877e1f0       get:            table[1]:917
Thread  0x877e1f0       release:        table[1] is released
Thread  0x8784268       alloc:          table[1]:632
Thread  0x8784268       get:            table[1]:632
Thread  0x8784268       release:        table[1] is released
Thread  0x879c448       alloc:          table[1]:147
Thread  0x879c448       get:            table[1]:147
Thread  0x879c448       release:        table[1] is released
Thread  0x87a24c0       alloc:          table[1]:890
Thread  0x87a24c0       get:            table[1]:890
Thread  0x87a8538       alloc:          alloc fail
Thread  0x8778178       get:            table[0]:53
Thread  0x8778178       release:        table[0] is released
Thread  0x878a2e0       alloc:          table[0]:146
Thread  0x878a2e0       get:            table[0]:146
Thread  0x878a2e0       release:        table[0] is released
Thread  0x8790358       alloc:          table[0]:841
Thread  0x8790358       get:            table[0]:841
Thread  0x8790358       release:        table[0] is released
Thread  0x87a24c0       release:        table[1] is released
Thread  0x87963d0       alloc:          table[0]:631
Thread  0x87963d0       get:            table[0]:631
Thread  0x87963d0       release:        table[0] is released
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

图 7: 线程数较少时 Table 的测试结果（Semaphore 实现）

有 10 个并发线程且有 2 个表槽时的运行情况，测试结果如图7所示。同样是 10 个并发线程，表槽数为 2 个，发现可以正常运行。

有 10 个并发线程但只有一个表槽时的运行情况，测试结果如图8所示。运行结果也与预期一致。

2.5 实现一个大小受限的缓冲区

1. 实验要求与分析

这个问题实际上是“生产者 - 消费者”问题。生产者消费者问题除各进程间需满足同步和互斥之外，还需注意生产者在缓冲区已满时不能再生产数据，消费者在缓冲区为空时不能消耗数据。这就要求生产者在缓冲区满时必须阻塞，等到有消费者消耗数据时唤醒生产者；消费者在缓冲区空时必须阻塞，等到有生产者消耗数据时唤醒消费者。为实现此目标，可以使用 Semaphore 机制，引入三个信号量分别用来控制互斥、缓冲区为空、缓冲区满三种情况。或者使用锁机制配合条件变量，引入一个锁控制互斥，两个条件变量控制缓冲区空或满。本实验中使用前者实现。

2. 代码实现

（1） **对原 BoundedBuffer.h 进行更正修改：**

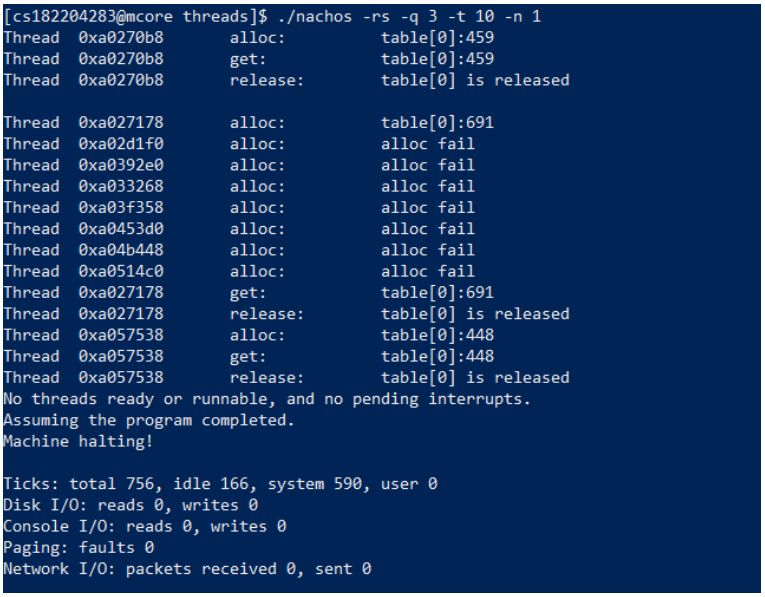

图 8: 线程数较多时 Table 的测试结果（Semaphore 实现）

思路描述：

添加引用头文件声明，完善原构造函数声明，添加构析函数声明，并添加私有变量 buffersize 表示缓冲区的大小，缓冲区指针 buffer，读和写的索引指针 readLoc 和 writeLoc，以及实现互斥和有限缓冲区的三个信号量 mutex，sEmpty，sFull：

```
#include "synch.h"
#include "system.h"
class BoundedBuffer {
public:
    // create a bounded buffer with a limit of 'maxsize' bytes
    BoundedBuffer(int maxsize);
    ~BoundedBuffer();

    // read 'size' bytes from the bounded buffer, storing into 'data'.
    // ('size' may be greater than 'maxsize')
    void Read(char *data, int size);

    // write 'size' bytes from 'data' into the bounded buffer.
    // ('size' may be greater than 'maxsize')
    void Write(char *data, int size);

    void printBuffer();

private:
    int bufferSize;
    char *buffer;
    char *readLoc, *writeLoc;

    Semaphore *sEmpty;
```

```
    Semaphore *sFull;
    Semaphore *mutex;
};
```

（2） **构造函数** `BoundedBuffer::BoundedBuffer()`

思路描述：

构造函数设置各成员变量的初值：根据传入参数设置缓冲区大小，并为缓冲区分配空间及分配初始值，设置三个信号量，初始化 readLoc、writeLoc 指针位置为缓冲区起始位置。

```
BoundedBuffer::BoundedBuffer(int maxsize) {
    bufferSize = maxsize;
    buffer = new char[bufferSize];
    sEmpty = new Semaphore("empty places", bufferSize);
    sFull = new Semaphore("used places", 0);
    mutex = new Semaphore("mutex of buffer", 1);
    for (int i = 0; i < bufferSize; i++) {
        buffer[i] = 0;
    }
    readLoc = writeLoc = buffer;
}
```

（3） **析构函数** `BoundedBuffer::~BoundedBuffer()`

思路描述：

构析函数释放构造函数所申请的所有空间及信号量：

```
BoundedBuffer::~BoundedBuffer() {
    delete sEmpty;
    delete sFull;
    delete mutex;
    delete buffer;
}
```

（4） `BoundedBuffer::Read(char *data, int size)`

思路描述：

用于消费者消耗数据。使用信号量 sFull 等待缓冲区有资源可以消耗，然后通过 mutex 来保证 read 时不会有其他线程访问缓冲区，实现线程互斥。在完成 read 后，解开互斥，并通知生产者缓冲区不为空：

```
void BoundedBuffer::Read(char *data, int size) {
    for (int i = 0; i < size; i++) {
```

```
        sFull->P();      //等待缓冲区不空，可以进行读数据
        mutex->P();      //保证互斥
        /*临界区*/
        data[i] = *readLoc;
        *readLoc = 0;    //消耗缓冲区数据
        ++readLoc;       //读指针后移
        if (readLoc == buffer + bufferSize) {    //循环读取数据
            readLoc = buffer;
        }
        printf("Reader (Thread %p): %c\n", currentThread, data[i]);
                         //输出当前线程信息
        printBuffer();

        mutex->V();      //退出临界区，释放资源
        sEmpty->V();     //此时缓冲区不满，发送信号量
    }
    data[size] = 0;
}
```

(5) `BoundedBuffer::Write(char *data, int size)`

思路描述：

用于生产者生产数据。使用信号量 sEmpty 等待缓冲区有位置可以写入，然后通过 mutex 来保证 read 时不会有其他线程访问缓冲区，实现线程互斥。在完成 write 后，解开互斥，并通知消费者缓冲区不为空：

```
void BoundedBuffer::Write(char *data, int size) {
    for (int i = 0; i < size; i++) {
        sEmpty->P();             //等待缓冲区不满，可以进行写数据
        mutex->P();              //保证互斥
        /*临界区*/
        *writeLoc = data[i];     //向缓冲区写入数据
        ++writeLoc;              //写指针后移
        if (writeLoc == buffer + bufferSize) {      //循环写数据
            writeLoc = buffer;
        }
        printf("Writer (Thread %p): %c\n", currentThread, data[i]);
                                 //输出当前线程信息
        printBuffer();

        mutex->V();              //退出临界区，释放资源
        sFull->V();              //此时缓冲区不空，发送信号量
    }
}
```

(6) `BoundedBuffer::printBuffer()`

思路描述：

用于输出缓冲区内容。

```
void BoundedBuffer::printBuffer() {
    printf("buffer:");
    for (int i = 0; i < bufferSize; i++) {
        printf("%c", buffer[i] ? buffer[i] : ' ');
    }
    printf("\n");
}
```

3. 实验测试

为了测试在多个线程随意切换的情况下是否能实现缓冲区安全，测试函数中开辟缓冲区，并 fork 若干线程。每个线程执行时，生产者每次写入一个数字（数字从 0 到 9 循环生成），而消费者每次读出一个数字：

3.1 测试函数：

```
char readBuffer[5], writeBuffer[5];
void BoundedBufferThread(int t) {
    if (t & 1) {
        writeBuffer[0] = Random() % 10 + '0';
        buffer->Write(writeBuffer, 1);
    }
    else {
        buffer->Read(readBuffer, 1);
    }
}
void ThreadTest4() {
    DEBUG('t', "Entering ThreadTest4");
    RandomInit(unsigned(time(0)));
    buffer = new BoundedBuffer(2);
    for (int i = 0; i < threadNum; i++) {
        Thread *t = new Thread("forker thread");
        t->Fork(BoundedBufferThread, i);
    }
    BoundedBufferThread(threadNum);
}
```

3.2 测试结果：

```
./nachos -rs -q 4 -t 10
```

（1） 设置缓冲区大小为 2，结果如图9所示：

```
buffer = new BoundedBuffer(2);
```

```
[cs182204133@mcore threads]$ ./nachos -rs -q 4 -t 10
Writer (Thread 0x8589220): 7
buffer:7
Writer (Thread 0x8595310): 2
buffer:72
Reader (Thread 0x859b388): 7
buffer: 2
Writer (Thread 0x85a1400): 7
buffer:72
Reader (Thread 0x85a7478): 2
buffer:7
Reader (Thread 0x858f298): 7
buffer:
Writer (Thread 0x85ad4f0): 6
buffer: 6
Writer (Thread 0x85b95e0): 6
buffer:66
Reader (Thread 0x85b3568): 6
buffer:6
Reader (Thread 0x85830b8): 6
buffer:

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 662, idle 22, system 640, 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...
```

图 9: BoundedBuffer 测试结果（缓冲区大小为 2）

（2） 设置缓冲区大小为 3，结果如图10所示：

```
buffer = new BoundedBuffer(3);
```

可见缓冲区的大小受限，且对于读写均安全，不会产生生产者产生数据溢出，或消费者在无数据时读取数据。

3 实验总结

本次实验重点在于探索线程同步与互斥，主要内容为实现信号量、锁机制和条件变量。其难点在于：

- 需要充分思考和理解其运行机制，设计条件变量中应有的变量及类型；
- 使用 `isHeldByCurrentThread()` 进行持有锁的判断，避免线程操作与锁状态冲突；
- 思考开关中断的时机和其必要性；
- 思考每条语句的先后执行顺序及其影响。

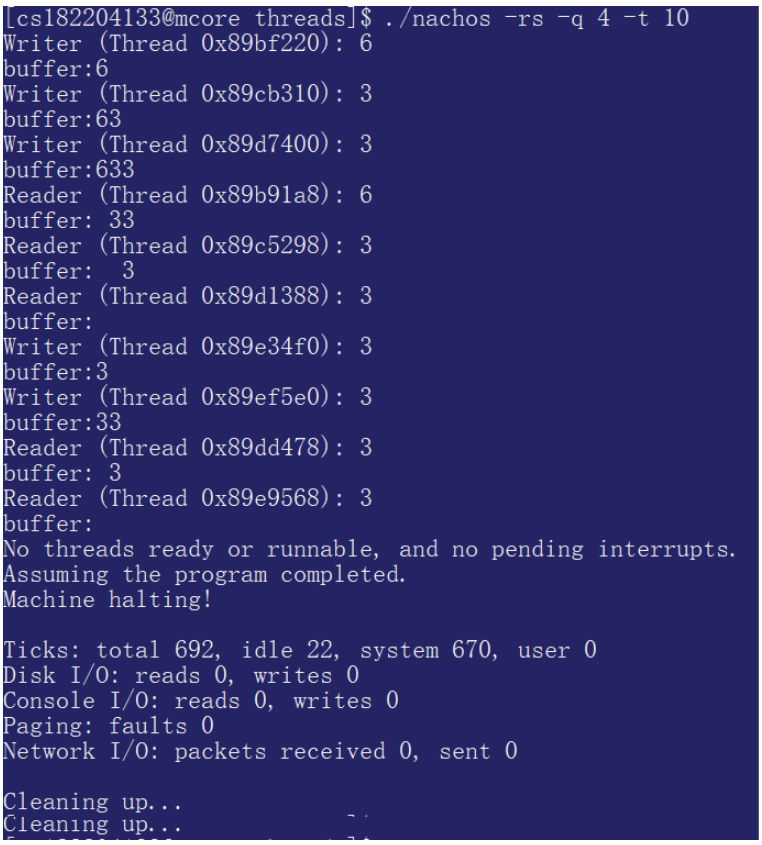

图 10: BoundedBuffer 测试结果（缓冲区大小为 3）

- 利用这些同步机制实现线程安全的数据结构操作，并在实现表、缓冲区的过程中思考同步机制设计是否完善，通过实验现象，找寻代码实现中的漏洞，检测其正确性与安全性。

A 小组分工情况

任务	代码实现、测试及相应报告部分
用 Sleep 实现锁和条件变量	吴朱冠宇
用 Semaphore 实现锁和条件变量	胡子潇
修改双向链表	吴朱冠宇、胡子潇
实现线程安全的表结构	谭心怡
实现一个大小受限的缓冲区	卞珂