操作系统第二次实验

姓名：李哲彦
学号：37220222203675

实现代码

Sleep 实现

首先考虑 Lock 类的实现。头文件定义如下：

```C++
class Lock {
 ... ...
  private:
    char* name;               // for debugging
    Thread *lockOwner;           // thread holding the lock
    List *lockQueue;             // threads waiting for the lock
    bool isLocked;           // true if the lock is held
    // plus some other stuff you'll need to define
};
```

这三个我自己新增的变量分别用于：判断当前锁的拥有者（是否本线程拥有锁），正在等待 (Acquire) 本锁的线程队列。最后一个用于判断当前锁是否已经有线程占用。

在析构函数中，要注意删除队列中的所有线程，不能只删除队列头。

```C++
while(!lockQueue->IsEmpty()){
      Thread *thread = (Thread *)lockQueue->Remove();
      scheduler->ReadyToRun(thread);  // 或标记为错误状态
    }
```

函数判断是否锁被当前线程拥有：较为简单，直接对比 lockOnwer 即可（在别的函数中也要注意对该变量的维护）。

```C++
    if(this->lockOwner == currentThread){
        return true;
    }
    else return false;
```

对于获取锁的操作，首先循环判断当前锁是否空闲（Mesa语义）。如果不空闲，就将当前线程加入等待队列，并睡眠。

```C++
while(this->isLocked){ // Unitl the lock is free, using while as we need Mesa
        lockQueue->Append((void *)currentThread);   // so go to sleep
        currentThread->Sleep();
    }
    this->isLocked = true;  // When the lock goes to free
    locOwner = currentThread; // the current thread is the owner of the lock
```

释放锁的操作：首先要判断当前线程是否拥有这个锁。如果不拥有，就不进行任何操作。否则先释放锁，然后激活一个等待线程（如果有的话）。

注意，此处存在一个易错点：必须先将锁释放，再激活新的线程，不然会造成活锁的错误。即先激活线程，再释放锁，会有可能导致该线程被激活后，发现锁依然处于 busy 状态，导致继续睡眠。

```C++
if(Lock::isHeldByCurrentThread()){ // Other threads are not permitted to release the lock
        this->isLocked = false; // Release 一定要在释放新线程之前释放，否则会导致活锁
        Thread *thread = (Thread *)lockQueue->Remove();
        if(thread != NULL){ // If there are hreads waiting for the lock
            scheduler->ReadyToRun(thread);
       }
        lockOwner = NULL;
    }
```

接下来是对条件变量的定义：

```C++
private:
    char* name;
    int signal; // 因为涉及到 broadcast，需要能够多次累积 signal
    List *waitQueue;  // Threads waiting on this condition
    // plus some other stuff you'll need to define
```

首先根据条件变量量定义，用一个 int 来表示条件变量量的值，可以理解为：当前还有多少个线程可以获取该条件变量。
然后还要维护一个队列，内容是正在等待该条件变量的线程。

等待信号量操作：

```C++
conditionLock->Release();
    this->signal--; // Use a signal
    while(this->signal < 0){ // Wait until the signal is == 0
        waitQueue->Append((void *)currentThread);
        currentThread->Sleep();
    }
    conditionLock->Acquire();  // Get the lock again
```

首先判断当前线程是否拥有对应的锁。然后，释放锁，并使用掉一个条件量，若小于 0 则进入等待状态。被唤醒后重新获取信号锁。

对于条件变量的 signal 函数，如果当前线程拥有锁，则先释放锁，然后就可以进行 signal 操作。同时要注意，必须判断等待队列中是否有线程正在等待，如果有再 ++ signal。否则会造成 Signal 操作累计。

```C++
conditionLock->Release(); // Release the lock first
if(waitQueue->IsEmpty() == false){
        this->signal++;  // signal 操作不会累计，只有存在 waiting 线程的才会加一
        Thread *thread = (Thread *)waitQueue->Remove();
        scheduler->ReadyToRun(thread);
    }
```

Broadcast 在此基础上略微修改即可，将 if 改为 while，不断释放直到完全释放完整个队列即可。

```C++
 while(waitQueue->IsEmpty() == false)
```

使用 Semaphore 实现

使用 Semaphore 信号量实现，无需自己维护变量。在原先基础上额外维护一个信号量即可。

```C++
private:
    Semaphore *s;
```

在这种情况下，条件变量的头文件定义无需进行改动。但是此处存在一个问题，即信号量的 V 操作是可以累计的，但我们要求条件变量的 Signal 操作不可累计。因此我选择通过 while 循环来消除累计的效果。

如下代码所示。虽然在 Signal 操作中，会不断累积计数。但由于我在 Wait 中使用了 while 操作，直到 Semaphore 信号量减到 0 以下才“罢休”，这样就与之前使用 Lock 且条件变量初始值为 0 的情况相同了。

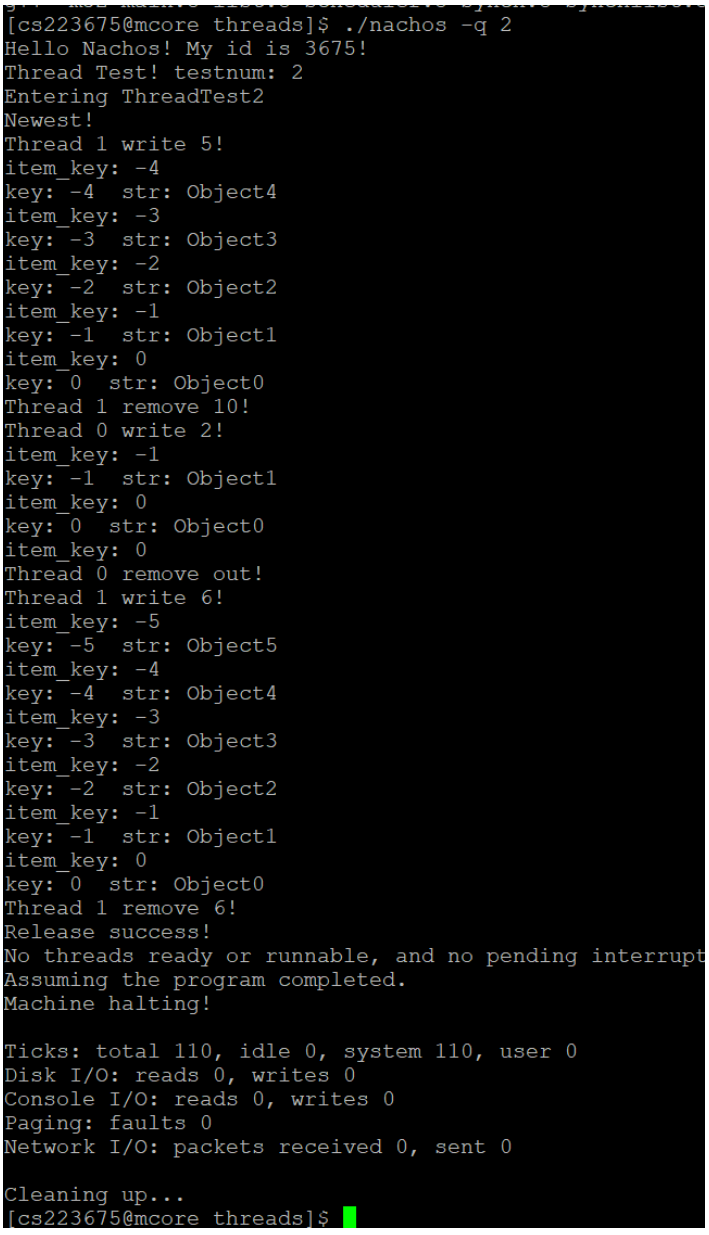
```C++
// Wait
    while(!s->P());   // 不断执行，直到其进入等待状态
// Signal
       s->V();   // 虽然会在 Semaphore 中累计，但由于 Wait 部分有 while 循环，可以消除这个累计
// BroadCast
          while(s->V());  // 不断释放，直到释放完为止
```

线程安全的链表

```C++
void DLListTest(int which){
    lock->Acquire();
    ...
    condition->Wait(lock);    // 等待信号再继续执行
    lock->Release();
```

```C++
    printf("Release success!\n");
    return ;
}
void DLListTest2(int which){
    lock->Acquire();
    currentThread->Yield();  // 强制让另一个线程先执行
    ...
    condition->Signal(lock); // 通知另一个线程继续执行
    lock->Release();
    return ;
}
```

使用 lock 来确保对链表的操作线程安全，同时使用条件变量来切换线程，最后得到的运行结果如下，并没有造成线程间数据混乱（顺便也验证了 Semaphore 实现的条件变量是正确的）



线程安全的Table 实现

对于 Table，很容易理解其对于线程安全的需求：同一时间只能有一个线程在修改列表，且在读取该列表的时候，不能有线程在修改当前列表。因此，使用一个锁直接控制所有 Table 的操作是最简单的选择。

以相对较简短的 Get 操作为例。一个 Get 操作不应当被打断，故关闭中断。并且，为保证线程安全，要获取到本 Table 专属的表后才能采取行动。

```C++
void* Table::Get(int index){
    if(index >= MaxSize) return NULL;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    lock->Acquire();
    struct node *now = head;
    while(now != NULL){
        ... do something
        now = now->next;
    }
    lock->Release();
    (void) interrupt->SetLevel(oldLevel);
    return NULL;
}
```

这也引出了在 Table.h 中的相关变量定义。在 Table 中，我们定义了一个 Lock 类变量，专门用于控制本 Table 的线程安全。在任何 Table 的操作函数时，必须要先获取到该线程锁，才能执行具体内容。

```C++
private:
      int MaxSize;
      struct node{
        链表操作定义
      } *head;
      int nowSize;
      Lock *lock;
```

以较为简短的 Get 操作为例：（具体操作请见注释）

```C++
void* Table::Get(int index){
    if(index >= MaxSize) return NULL;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);    // 原子操作，关中断
    lock->Acquire();    // 获取线程锁
    struct node *now = head;    // 常规列表操作
    while(now != NULL){
        if(now->index == index){
            lock->Release();
            return now->value;
        }
        now = now->next;
    }
    lock->Release();   // 释放锁
    (void) interrupt->SetLevel(oldLevel);   // 重新打开中断
    return NULL;
}
```

其余两个操作与 Get 操作类似。每次要确保先获取线程锁，才能进行后续操作，从而确保了不会有多个线程在同时访问或修改该 Table。

在 ThreadTest 中测试该 Table 的功能，同时创建了多个线程进行不同的操作。最后，Table 运行结果正常，结果符合线程获取到锁之后的运行兴趣。

```C++
void ThreadTest4(){
    DEBUG('t', "Entering ThreadTest4");
    Thread *t = new Thread("Get");
    Thread *t1 = new Thread("Get");
    Thread *t2 = new Thread("Release");
    Thread *t3 = new Thread("Write");
    t3->Fork(TableWrite, 2);
    t2->Fork(TableRelease, 0);
    t->Fork(TableGet, 0);
    t1->Fork(TableGet, 1);
    TableWrite(1);
    return ;
}
```

运行结果:

```
[cs223675@mcore threads]$ ./nachos -q 4
Hello Nachos! My id is 3675!
Thread Test! testnum: 4
Threadtest4
Write
Write
Object1
Get
Object2
Get
Release
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, 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...
```

安全的 BoundedBuffer

首先看 Buffer 的定义。对于缓冲区本体，我们需要有头指针以及 in, out 首尾指针来确定。由于这涉及到生产者与消费者问题，再使用三个 Semaphore 信号量来实现（为了尽量少引入额外的变量，没有选择信号量与 Lock 的实现）。最后，还要考虑到，传统的生产-消费问题一次只引入一个单位的内容。但是，此处我们要求实现连续获取/生产多个单位内容。为防止写入与读出的顺序混乱（即同时也多个生产者在写入，或同时也多个消费者在读出），使用了两把锁来对生产者和消费者进行控制。

```C++
private:
    void *buffer; // buffer 指针
    int in, out; // 缓冲区范围指针
    Semaphore *count;  // 表示缓冲区有多少内容
    Semaphore *getBuffer; // 获取权限
    Semaphore *e;  // 表示剩下多少容量
    int maxSize;
    Lock *Producer;
    Lock *Customer;   // 防止同时有多个生产者和消费者，造成写入和读取顺序混乱
```

接下来是其具体实现。

以 Read 操作为例。由于我们读出多个数据，因此要使用循环进行多次读出尝试，每次循环只能读出一个单位数据。在每次读取前，要先获取 count 的信号量，表示缓冲区至少还有一个单位内容，然后获取 getBuffer 信号量，这一信号量主要用于控制生产者与消费者的线程间安全，可以用 Lock 代替。

消费者每次执行完之后，执行一次 e 的 V 操作，将缓冲区剩余容量加一。（其中，e 信号量初始化为缓冲区的最大容量 `this->e = new Semaphore("eSem", maxsize);` ）

```C++
void BoundedBuffer::Read(void *data, int size){
    int now = 0;
    IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts
    this->Customer->Acquire(); // 屏蔽其他消费者
    while(now < size){
        this->count->P();  // 只能保证至少含有一个字节，故需要拆成多次循环
        this->getBuffer->P();
        memcpy((void*)((char*)data + now), (void*)((char*)this->buffer + out), 1); // 按照单字节来操作
        out++;
        now++;
        if(out == maxSize) out = 0;
        this->getBuffer->V();
        this->e->V();
    }
    this->Customer->Release();
    (void) interrupt->SetLevel(oldLevel);   // re-enable interrupts
    return ;
}
```

Write 操作类似，不再赘述。

对 Bounded Buffer 的测试:

首先，定义一个初始化容量只有 5 的缓冲区。较小的缓冲区可用于测试生产者是否会在缓冲区容量满时，根据 e 信号量及时等待。

```C++
BoundedBuffer* buffer = new BoundedBuffer(5); // 5 字节容量
```

之后，定义两个写入和读出操作，每次操作的内容长度至少为 7 字节以上，以测试缓冲区在超出其容量的情况下的运行情况。

```C++
void ThreadTest3(){   // 定义了两个写入和读出操作
    DEBUG('t', "Entering ThreadTest3");
    Thread *t = new Thread("forked thread");
    Thread *t1 = new Thread("write");
    Thread *t2 = new Thread("Read");
    t2->Fork(BufferWrite, 1);
    t->Fork(BufferWrite, 2);
    t1->Fork(BufferRead, 3);
    BufferRead(4);
    return ;
}
```

运行结果：

```
[cs223675@mcore threads]$ ./nachos -q 3
Hello Nachos! My id is 3675!
Thread Test! testnum: 3
Entering ThreadTEst3
Thread 4 read out!
Object1
Thread 3 read out!
Object2
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 110, idle 0, system 110, 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...
[cs223675@mcore threads]$
```

可以看到两个字符串都被正确读出，这两个字符串长度均大于缓冲区长度。可见信号量以及 Buffer 工作正常。

遇到的问题

- 问题1： 活锁问题。在最开始实现 Lock 类的时候，我并没注意在 Release 操作中的活锁问题，而是先激活队列中的等待线程，再修改锁的状态。最后发现，在进行 ThreadTest 测试时，部分情况下的 Lock 类不能正确工作，一部分线程并没有按照预想的情况重新继续执行完毕。经过排查，发现是活锁问题，修复后错误消失。
- 问题2：生产者与消费者问题的理解与扩展：在课本上的生产者-消费者问题中，每次只写入一个单位的内容，因此无需考虑多个消费者与生产者同时工作的问题。为了确保读取、

写入的顺序连续，额外创建了两个锁，每次进行写入或读取前，要获取相应的锁，从而屏蔽其他生产者与消费者。

- 问题3：使用信号量来实现条件变量时，如何消除信号量 V 操作累计的问题。为解决该问题，我略微修改了 Semaphore 的函数返回值，如果其成功进入了等待状态，才会退出 while 循环。这样，就可以避免信号量 V 操作带来的累计问题。

感想：

1. **同步机制的理解深化：** 通过手动实现锁和条件变量，深刻理解了Mesa语义与Hoare语义的区别，了解了条件变量与信号量的区别。条件变量的"信号即提示"而非"信号即占有"特性，将二者进行了区分。

2. 对消费者与生产者问题的进一步理解。虽然课本上对基础的消费者-生产者问题有了讲解，但是本实验我成功对其进行了自主修改，掌握了三个信号量的意义与使用。

3. 对信号量、锁以及条件变量三者的概念区分以及具体实现方法有了更深刻的认知。在未来的学习中，要好好利用这三者来确保线程安全。