同步机制实习报告

姓名 **彭广举** 学号 1200012964
日期 2015-3-31

目录

内容一：总体概述

本次 lab 主要是学习 nachos 的同步机制，并且根据已学过的知识对其同步机制进行扩展。

Synch.h/synch.cc 是对 nachos 的同步机制进行管理的，已有了对于信号量 PV 操作的实现，然后根据 synchlist 中对于对 Lock 和 Condition Variable 的使用，以及操作系统的知识，补全对 Lock 和 Condition Variable 的实现。

然后，构造一个经典的 IPC 问题，例如生产者消费者问题，来测试自己补全的同步机制。

最后，可以挑战实现 Linux 中有的 Barrier 和读写锁的机制。

【用简洁的语言描述本次 lab 的主要内容；阐述本次 lab 中涉及到的重要的概念，技术，原理等，以及其他你认为的最重要的知识点。这一部分主要是看大家对 lab 的总体的理解。

要求：简洁，不需要面面俱到，把重要的知识点阐述清楚即可。】

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Challenge1	Challenge2
第一部分	Y	Y	Y	Y	Y

具体 Exercise 的完成情况

第一部分

Exercise1

调研 **Linux** 中实现的同步机制。

1. 原子操作：不可分割的，在执行完之前不会被其他任务或事件中断，即是最小的执行单位。原子操作需要硬件支持，因此是架构相关的，其 API 和原子类型的定义都在内核源码树的 include/asm/atomic.h 文件中，使用汇编编写。原子操作主要实现资源计数。

2. 自旋锁：为了防止多处理器并发而引入的一种锁，在内核中大量应用于中断处理等部分。自旋锁最多只能被一个内核任务所持有，如果一个内核任务试图请求一个已被占用的自旋锁，这个任务会一直忙等待等待锁重新可用。自旋锁可以在任何时刻防止多于一个的内核任务同时进入临界区，因此这种锁可以有效地避免多处理器上并发运行的内核任务竞争共享资源。

3. 信号量：信号量在创建时需要设置一个初值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初值为 1 就变为互斥锁，即同时只能有一个任务可以访问信号量保护的共享资源。一个任务想要访问共享资源，首先必须得到信号量，获取信号量的操作把信号量的值减 1,若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加 1 实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

4. 读写锁：如果每个执行实体对临界区的访问或者是读或者是写共享数据结构，但是它们都不会同时进行读和写操作，读写锁是最好的选择。

5. 完成变量：如果内核中的一个进程需要发出信号通知另一个进程发生了某特定事件，利用完成变量是可以使两个进程得以同步的简单方法。

6. 顺序锁：顺序锁用于读写共享数据，这种锁依靠一个序列计数器，写时会得到一个锁，序列值增加，读时，读前读后都读取序列值，若相同则成功读。适用于读者多，写者少，希望写优先于读。

7. 屏障：用于对一组线程进行协调，一组线程协同完成一项任务，需要所有线程都到达一个会合点后再一起向前推进。

Exercise2

源代码阅读：**synch.h/synch.cc, synchlist.h/synchlist.cc**

synch.h/synch.cc 定义了三种同步机制的类：信号量 Semaphore，锁 Lock 和条件变量 Condition。在同步互斥机制的实现中，很多操作都是原子操作。Nachos 是运行在

单一处理器上的操作系统，在单一处理器上，实现原子操作只需要在操作之前关中断，操作之后恢复原来中断状态即可。

Semaphore 定义了两种操作 P 和 V，并且用一个 List 链表来保存等待在该信号量上的线程。P 操作：当 value 值等于 0, 将当前运行线程放入线程等待队列，当前运行线程进入睡眠状态，并切换到其他线程运行；当 value 值大于 0 时，value--。V 操作：如果线程等待队列中有等待该信号量的线程，取出其中一个将其设置为就绪态，放入就绪队列，准备运行，value++。在执行 PV 操作之前都需要先关闭中断以实现原子操作，结束时，恢复之前中断状态。在 P 操作中对 value==0 的判断用 while 而不是 if，考虑下面情况，假设已有若干线程等待在信号量 S1 上，这时正在运行的线程调用 S1 的 V 操作将一个等待线程 t1 放入就绪队列，并使得 value++，但此时 t1,并没有运行。时间片到来或当前线程主动 Yield，或从就绪队列中选取一个线程运行，可能是 t2,如果 t2 也调用了 S1 的 P 才哦做，那么 value 值又变为了 0,若用 if 的话，当 t1 被调度时，会执行 value--，这样 value 变成了负数而出错。

Lock，锁有两种状态 BUSY 和 FREE。当锁处于 FREE 态时，线程可以取得该锁后进入临界区，执行完临界区操作之后，释放锁；当锁处于 BUSY 态时，需要申请该锁的线程进入睡眠状态，等锁为 FREE 态时，再取得该锁。锁有两个原子操作 Acquire 和 Release 用于获取和释放锁。锁相当于二值信号量，可用信号量的 PV 操作实现。

Condition，条件变量和信号量与锁的机制不一样，它是没有值的。当一个线程需要的某种条件没有得到满足事时，可以将自己作为一个等待条件变量的线程插入所有等待该条件变量的队列，只要条件一旦满足，该线程就会被唤醒继续运行。条件变量有三个操作 Wait，Signal 以及 BroadCast，所有这些操作必须在当前线程获得一个锁的前提下，而且所有对一个条件变量进行的操作必须建立在同一个锁的前提下。

Synchlist.h/synchlist.cc 定义了一个同步链表的数据结构，用锁和条件变量实现，有两个主要的操作 Append 和 Remove。Append 操作在链表的尾部追加一个元素，并且唤醒等在该元素的线程；Remove 操作从链表头部取一个元素，如果链表空，则等待。

Exercise3

实现锁和条件变量

锁的实现:

可以用二值条件变量直接实现，我这里仍旧采用开关中断的方法实现原子操作。

Lock 类新加变量:

```
Thread* lockThread;          // 持有该锁的线程
int value;                   // 初值为 1
List *queue;                 // 保存等待在该锁上的线程
```

分别扩展 Acquire/Release/isHeldByCurrentThread 函数:

isHeldByCurrentThread：只需判断当前线程是否是 lockThread，若是则返回 true，否则返回 false。

Acquire 获取锁：类似于 P 操作，先关中断，然后判断 value 是否为 0,若是，则将当前线程加入该锁的队列 queue，并睡眠；若不是 0,则执行 value--，并将当前线程的指针赋给 lockThread，最后恢复中断。

Release 释放锁：同样先关中断，然后判断该线程是否持有该锁，若否，终止退出，若是，将 lockThread 设为 NULL，并从队列 queue 中取出等待的线程 thread，若 thread 不是 NULL，将其放入就绪队列，value++，最后恢复中断。

条件变量的实现：

Condition 类新增变量：

List *queue;　　// 保存等待在该条件变量上的线程

需要对 Wait，Signal 和 Broadcast 函数进行扩充，在 Nachos 中，这三个操作都必须在当前线程获得一个锁的前提下进行，而且所有对于一个条件变量的操作必须建立在同一个锁的前提下。为了保证操作的原子性，依然使用开关中断。

Wait：将线程等待在该条件变量上。首先关中断，判断当前线程是否持有参数中的锁，若否，则终止退出，否则释放该锁，然后将当前线程追加到条件变量的等待队列 queue 中，并睡眠。之后，当线程重新调度时，重新获取锁，最后恢复中断。

Signal：唤醒一个等待的线程。关中断，确保当前线程持有锁，之后从队列 queue 中移出一个线程 thread，若 thread 不为 NULL，将其放入就绪队列，等待被调度，符合 Mesa 管程的语义，最后恢复中断。

Broadcast：唤醒所有等待在该条件变量上的线程。同样关中断，确保当前线程持有锁，遍历链表 queue，依次执行 Signal 操作，直到队列为空，最后恢复中断。

Exercise4

实现同步互斥实例

分别用信号量和条件变量实现了生产者消费者问题，对固定大小为 5 的 buffer 数组进行操作。

1. 做为对比，先看一下没有用同步机制的生产这消费者：

```
void Producer(int which)
{
    int i;
    for (i = 0; i < MaxOps; ++i)
    {
        if (empty_count > 0)
        {
            full_count++;
            empty_count--;
            buffer[in] = i+which*MaxOps;
            interrupt->OneTick();  //模拟时钟移动
             printf("*** thread %d Produce an Item %d, Full: %d, Empty: %d\n", which, i+which*MaxOps, full_count, empty_count);
            interrupt->OneTick();  //模拟时钟移动
            currentThread->Yield(); // 为了促成可能的错误
            in = (in + 1) % BufferNum;
            if (i % 2 == 1)
```

```
            currentThread->Yield();
        }
        else
        {
          i--;
          interrupt->OneTick();  //模拟时钟移动
        }
    }
}

void Consumer(int which)
{
    int i;
    for (i = 0; i < MaxOps*2; ++i)
    {
        if (full_count > 0)
        {
            full_count--;
          empty_count++;
            int val = buffer[out];
            buffer[out] = -1;
            interrupt->OneTick();  //模拟时钟移动
             printf("*** thread %d Consume an Item %d, Full: %d, Empty:
%d\n", which, val, full_count, empty_count);
            interrupt->OneTick();  //模拟时钟移动
            out = (out + 1) % BufferNum;
        }
        else
        {
            i--;
            interrupt->OneTick();  //模拟时钟移动
        }
    }
}
```

为了更容易地出现同步错误，在生产者的 in 更新之前，调用了一次 Yield。

创建 3 个线程：两个生产者一个消费者，每个生产者执行 5 次，消费者执行 10 次，运行结果如下：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** thread 1 Produce an Item 5, Full: 1, Empty: 4
*** thread 2 Produce an Item 10, Full: 2, Empty: 3
*** thread 3 Consume an Item 10, Full: 1, Empty: 4
*** thread 3 Consume an Item -1, Full: 0, Empty: 5
Ticks: 200, TimeInterrept!
Ticks: 400, TimeInterrept!
*** thread 2 Produce an Item 11, Full: 1, Empty: 4
*** thread 1 Produce an Item 6, Full: 2, Empty: 3
The interval is to short!!!
*** thread 2 Produce an Item 12, Full: 3, Empty: 2
*** thread 2 Produce an Item 13, Full: 4, Empty: 1
*** thread 1 Produce an Item 7, Full: 5, Empty: 0
Ticks: 600, TimeInterrept!
*** thread 3 Consume an Item 6, Full: 4, Empty: 1
*** thread 3 Consume an Item -1, Full: 3, Empty: 2
*** thread 3 Consume an Item 12, Full: 2, Empty: 3
*** thread 3 Consume an Item 7, Full: 1, Empty: 4
*** thread 3 Consume an Item 11, Full: 0, Empty: 5
Ticks: 800, TimeInterrept!
The interval is to short!!!
*** thread 2 Produce an Item 14, Full: 1, Empty: 4
*** thread 1 Produce an Item 8, Full: 2, Empty: 3
*** thread 3 Consume an Item 14, Full: 1, Empty: 4
*** thread 3 Consume an Item 8, Full: 0, Empty: 5
Ticks: 1000, TimeInterrept!
Ticks: 1200, TimeInterrept!
*** thread 1 Produce an Item 9, Full: 1, Empty: 4
*** thread 3 Consume an Item 9, Full: 0, Empty: 5
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1400, idle 140, system 1260, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
```

发现 Thread3 在消费的时候出现了-1,这是由于 Thread1 写数据到缓冲区的第一个位置，但在 in 还未更新之前，切换到了 Thread2，Thread2 原本应该写数据到缓冲区第二个位置，但 in 未更新，所以仍然写进了第一个位置，第二个位置为-1，所以接下来消费者两次取得的数据分别为 10 和-1，第一次写进的 5 丢失了。

2. 用信号量实现生产者消费者问题

定义了三个信号量：

```
Semaphore *full = new Semaphore("FULL", 0);
Semaphore *empty = new Semaphore("EMPTY", 5);
Semaphore *mutex = new Semaphore("MUTEX", 1);
```

其中 mutex 实现互斥操作，full 和 empty 用于插入和删除，其初值分别为 0 和

5.

生产者：

```
void ProducerTest(int which)
{
   int i;
   for (i = 0; i < MaxOps; ++i)
   {
     empty->P();
     mutex->P();
     full_count++;
     empty_count--;
      buffer[in] = i+which*MaxOps;
      interrupt->OneTick();  //模拟时钟移动
         printf("*** thread %d Produce an Item %d, Full: %d, Empty: %d\n", which, i+which*MaxOps, full_count, empty_count);
      interrupt->OneTick();  //模拟时钟移动
      currentThread->Yield();  // 为了促成可能的错误
      in = (in + 1) % BufferNum;
     mutex->V();
     full->V();
      if (i % 2 == 1)
         currentThread->Yield();
   }
}
```

循环执行 MaxOps 次生产操作，用 OneTick 模拟时钟移动。先执行 empty 的 P 操作，确保有空位可供插入。插入数据时用互斥变量 mutex 做保护，最后执行 full 的 V 操作，说明可消费的数据增加 1，唤醒等待在该信号量上的线程。为了使线程切换更频繁，让生产者没生产 2 个 Yield 一次。

消费者：

```
void ConsumerTest(int which)
{
   int i;
   for (i = 0; i < MaxOps*2; ++i)
   {
     full->P();
     mutex->P();
     full_count--;
```

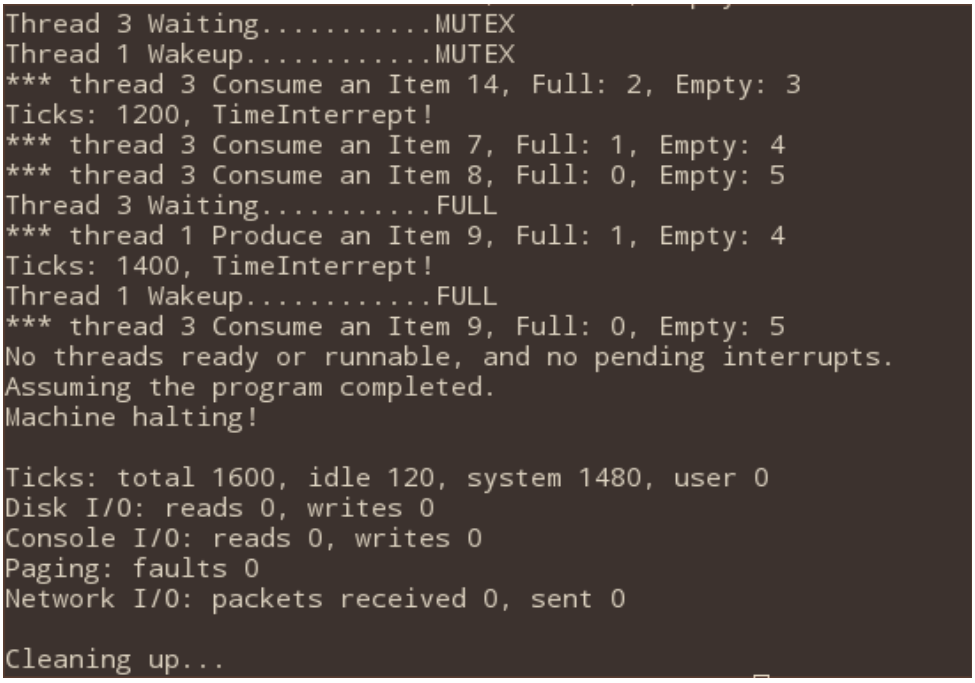
```
        empty_count++;
         int val = buffer[out];
         buffer[out] = -1;
         interrupt->OneTick();  //模拟时钟移动
           printf("*** thread %d Consume an Item %d, Full: %d, Empty:
    %d\n", which, val, full_count, empty_count);
         interrupt->OneTick();  //模拟时钟移动
         out = (out + 1) % BufferNum;
        mutex->V();
        empty->V();
      }
    }
```

循环执行 MaxOps*2 次消费操作。先执行 full 的 P 操作，确保有数据可以消费，然后用互斥 mutex 确保操作的安全性，最后执行 empty 的 V 操作，说明空位增加 1，唤醒等待空位的生产操作。

默认生产操作更耗时，所以执行了两次 OneTick。

定义 3 个分别叫 Producer1,Producer2 的 Consumer 的线程，MaxOps=5，运行结果如下：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** thread 1 Produce an Item 5, Full: 1, Empty: 4
Thread 2 Waiting...........MUTEX
Thread 3 Waiting...........FULL
Thread 1 Wakeup............MUTEX
Thread 1 Wakeup............FULL
*** thread 1 Produce an Item 6, Full: 2, Empty: 3
Thread 2 Waiting...........MUTEX
Ticks: 200, TimeInterrept!
The interval is to short!!!
Thread 3 Waiting...........MUTEX
Thread 1 Wakeup............MUTEX
*** thread 2 Produce an Item 10, Full: 3, Empty: 2
Thread 2 Wakeup............MUTEX
*** thread 2 Produce an Item 11, Full: 4, Empty: 1
*** thread 3 Consume an Item 5, Full: 3, Empty: 2
Ticks: 400, TimeInterrept!
*** thread 3 Consume an Item 6, Full: 2, Empty: 3
*** thread 3 Consume an Item 10, Full: 1, Empty: 4
*** thread 3 Consume an Item 11, Full: 0, Empty: 5
Ticks: 600, TimeInterrept!
Thread 3 Waiting...........FULL
*** thread 2 Produce an Item 12, Full: 1, Empty: 4
Thread 2 Wakeup............FULL
*** thread 2 Produce an Item 13, Full: 2, Empty: 3
Ticks: 800, TimeInterrept!
Thread 3 Waiting...........MUTEX
Thread 1 Waiting...........MUTEX
*** thread 2 Produce an Item 14, Full: 3, Empty: 2
Thread 2 Wakeup............MUTEX
*** thread 3 Consume an Item 12, Full: 2, Empty: 3
Thread 3 Wakeup............MUTEX
*** thread 3 Consume an Item 13, Full: 1, Empty: 4
Ticks: 1000, TimeInterrept!
*** thread 1 Produce an Item 7, Full: 2, Empty: 3
*** thread 1 Produce an Item 8, Full: 3, Empty: 2
```



对于 buffer 的操作和不用信号量的操作完全一样，但结果没有出现错误，说明用信号量的 PV 操作解决了生产者消费者的同步问题。

2. 条件变量实现生产者消费者问题

在 synch.h/synch.cc 中，定义一个新的类 Monitor_PC 类似于一个实现生产者消费者问题的管程。

```
class Monitor_PC {
    public:

      Monitor_PC(char* debugName);
     ~Monitor_PC();

      char* getName() { return (name); }
      void insert(int id, char* threadName, int item);
     int remove(int id, char* threadName);

    private:
      Condition* full;
      Condition* empty;
      Lock* Clock;
      char* name;
      int count;
```

```
    int m_buffer[MaxBuffer];
      int m_in;
      int m_out;
};
```

用条件变量 full 和 empty 来维护生产者消费者模型，count 用于计数。当 count==MaxBuffer 时，insert 操作在条件 full 上等待，当 count==1 时，唤醒等待在条件 empty 上的 remove 操作，当 count==0 时，remove 操作等待在 empty 条件上，当 count==MaxBuffer-1 时，唤醒等待在 full 上的 insert 操作。另外进入 insert 或者 remove 时均需要获取锁，这个锁也用于条件的 wait 和 signal，离开时，释放锁。

insert 和 remove 操作中对缓冲区的操作也类似于之前的两种方法中的操作。

测试函数就是简单地循环进行 insert 和 remove 操作，生产者仍然每两次 yield 一次，同样创建 3 个线程，结果如下：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
Thread 1 Insert.............
*** thread 1 name Producer1 Produce an Item 5, Count: 1
Thread 1 Wakeup.............EMPTY
Thread 1 Insert.............
*** thread 1 name Producer1 Produce an Item 6, Count: 2
The interval is to short!!!
Ticks: 200, TimeInterrept!
Thread 1 Insert.............
*** thread 1 name Producer1 Produce an Item 7, Count: 3
Thread 1 Insert.............
*** thread 1 name Producer1 Produce an Item 8, Count: 4
The interval is to short!!!
Thread 1 Insert.............
*** thread 1 name Producer1 Produce an Item 9, Count: 5
Thread 2 Insert.............
Thread 2 Waiting............FULL
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 5, Count: 4
Ticks: 400, TimeInterrept!
Thread 3 Wakeup.............FULL
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 6, Count: 3
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 7, Count: 2
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 8, Count: 1
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 9, Count: 0
Ticks: 600, TimeInterrept!
Thread 3 Remove.............
Thread 3 Waiting............EMPTY
*** thread 2 name Producer2 Produce an Item 10, Count: 1
Thread 2 Wakeup.............EMPTY
Thread 2 Insert.............
*** thread 2 name Producer2 Produce an Item 11, Count: 2
```

```
The interval is to short!!!
Thread 2 Insert.............
*** thread 2 name Producer2 Produce an Item 12, Count: 3
Thread 2 Insert.............
Ticks: 800, TimeInterrept!
*** thread 2 name Producer2 Produce an Item 13, Count: 4
*** thread 3 name Consumer Consume an Item 10, Count: 3
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 11, Count: 2
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 12, Count: 1
Thread 3 Remove.............
*** thread 3 name Consumer Consume an Item 13, Count: 0
Ticks: 1000, TimeInterrept!
Thread 3 Remove.............
Thread 3 Waiting.............EMPTY
Thread 2 Insert.............
*** thread 2 name Producer2 Produce an Item 14, Count: 1
Thread 2 Wakeup.............EMPTY
*** thread 3 name Consumer Consume an Item 14, Count: 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1200, idle 70, system 1130, 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...
```

同样符合预期，没有出现错误。

Challenge1:

实现 **Barrier**：可以使用 **Nachos** 提供的同步互斥机制（如条件变量）来实现 **barrier**，使得当且仅当若干个线程同时到达某一点时方可继续执行。

主要用锁和条件变量实现，在 synch.h/synch.cc 中增加类 Barrier，类中信息如下：

```
class Barrier {
    public:
      Barrier(char* debugName, int num);
      ~Barrier();

      char* getName() { return (name); }
      void setBarrier();  // 设置 Barrier
    private:
      char* name;
```

```
    int status;        // 0 表示该屏障有效，1 表示该屏障失效
    int waitNum;       // 需要等待线程的个数
    Lock* CLock;       // 对于 Barrier 状态的修改和条件变量的操作需要的锁
    Condition* bc;     // 等待在的条件变量
};
```

主要的操作只有一个 setBarrier，在线程中调用 setBarrier，则该线程等待在此处，直到有 waitNum 个线程等待在该 Barrier 上，要求 waitNum>0，否则终止退出。

```
void Barrier::setBarrier()
{
    CLock->Acquire();
    ASSERT(status == 0);    // 试图设置一个已失效的 Barrier，则终止
    waitNum--;
    if (waitNum == 0)
    {
      status = 1;          // 所有线程都到达了，设置该 Barrier 失效
      bc->Broadcast(CLock); // 将所有等在该 Barrier 处的线程唤醒
    }
    else
      bc->Wait(CLock);     // 线程数还不够，则等待
    CLock->Release();
}
```

使用时，新建一个 Barrier 类，指明需要等待的线程数目，然后在线程中调用该 Barrier 对象的 setBarrier 函数。

测试用了 5 个线程，只有 5 个线程都到达 Barrier 的设置点时，再向前推进，运行结果如下：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
Thread 1 Name Barrier1 Waiting Here!
Thread 2 Name Barrier2 Waiting Here!
Thread 3 Name Barrier3 Waiting Here!
Thread 4 Name Barrier4 Waiting Here!
Ticks: 200, TimeInterrept!
Thread 5 Name Barrier5 Waiting Here!
Thread 5 Name Barrier5 Continue to Run
Thread 1 Name Barrier1 Continue to Run
Thread 2 Name Barrier2 Continue to Run
Thread 3 Name Barrier3 Continue to Run
Ticks: 400, TimeInterrept!
Thread 4 Name Barrier4 Continue to Run
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 600, idle 170, system 430, 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...
```

如图当 5 个线程都到达 Barrier 的设置点时才向前推进。

Challenge2:

实现 **read/write lock**：

基于 **Nachos** 提供的 **lock(synch.h** 和 **synch.cc)**，实现 **read/write lock**。使得若干线程可以同时读取某共享数据区内的数据，但是在某一特定的时刻，只有一个线程可以向该共享数据区写入数据。

同样在 synch.h/synch.cc 中增加了一个类 ReadWriteLock。采取的读写策略是基于第三类读者写者问题，相对较公平。允许同时多个线程在读，只能有一个线程在写。但有线程正在读时，若先有写者到来，那么后续的读者等在写者之后，以减少写者饥饿的可能性。

实现如下：

类 ReadWriteLock：

```
class ReadWriteLock {
 public:
   ReadWriteLock(char* debugNum);
   ~ReadWriteLock();

   char* getName() { return (name); }
   void ReadLockAcquire();
   void ReadLockRelease();
   void WriteLockAcquire();
   void WriteLockRelease();
 private:
   char* name;
   Lock* mutex;       // 互斥访问，实现对 rCount 的互斥修改
   Lock* rLock;       // 主要用于读写线程的协调
   Lock* wLock;       // 互斥写
   int rCount;        // 记录正在读的线程数
};
```

如上面注释提到的使用了三个锁，以保证读者写者操作的互斥。实现的操作有对读锁的获取与释放和对写锁的获取与释放。

读锁的获取，首先需要获取 rLock，然后获取 mutex 确保对 rCount 的互斥访问，rCount++，若 rCount 为 1，说明这是该批读者的第一个，那么去申请获取写锁 wLock，接下来释放 mutex 和 rLock。读锁的释放，主要是对 rCount 的修改和对等待在锁 wLock 的写者的唤醒，同样用 mutex 保证对 rCount 操作的互斥，rCount--，若 rCount 为 0，释放写锁，以唤醒写线程。

写锁相对简单，获取时，先获取 rLock，保证这次写操作先于之后到来的读操作，然后获取写锁，保证只有一个线程在写，然后释放 rLock。释放时，只需释放 wLock 即可。

测试：

设置一个全局的共享数组 shared，有 3 个元素，初值均为 100。读操作，读取 3 个元素，写操作将 3 个元素的值都减 1。设置了 9 个线程，5 个读，4 个写，它们添加到就绪队列中的顺序为：rrwrwrwwr，这样的设置方便看出可同时读，只能一个写的状况。用若干 OneTick 推动时钟前进。

没有读写锁时的结果：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** Thread 1 Name Read1 Read Shared Array 0: 100
*** Thread 1 Name Read1 Read Shared Array 1: 100
Ticks: 200, TimeInterrept!
*** Thread 2 Name Read2 Read Shared Array 0: 100
*** Thread 2 Name Read2 Read Shared Array 1: 100
*** Thread 2 Name Read2 Read Shared Array 2: 100
*** Thread 3 Name Write1 Write Shared Array 0: 99
Ticks: 400, TimeInterrept!
*** Thread 4 Name Read3 Read Shared Array 0: 99
*** Thread 4 Name Read3 Read Shared Array 1: 100
*** Thread 4 Name Read3 Read Shared Array 2: 100
*** Thread 5 Name Write2 Write Shared Array 0: 98
Ticks: 600, TimeInterrept!
*** Thread 6 Name Read4 Read Shared Array 0: 98
*** Thread 6 Name Read4 Read Shared Array 1: 100
*** Thread 6 Name Read4 Read Shared Array 2: 100
*** Thread 7 Name Write3 Write Shared Array 0: 97
Ticks: 800, TimeInterrept!
*** Thread 8 Name Write4 Write Shared Array 0: 96
*** Thread 8 Name Write4 Write Shared Array 1: 99
*** Thread 8 Name Write4 Write Shared Array 2: 99
Ticks: 1000, TimeInterrept!
*** Thread 9 Name Read5 Read Shared Array 0: 96
*** Thread 9 Name Read5 Read Shared Array 1: 99
*** Thread 9 Name Read5 Read Shared Array 2: 99
Ticks: 1200, TimeInterrept!
*** Thread 5 Name Write2 Write Shared Array 1: 98
*** Thread 5 Name Write2 Write Shared Array 2: 98
Ticks: 1400, TimeInterrept!
The interval is to short!!!
*** Thread 7 Name Write3 Write Shared Array 1: 97
*** Thread 7 Name Write3 Write Shared Array 2: 97
Ticks: 1600, TimeInterrept!
```

如上结果，预期的同一个线程读到的值应该相同，但 Thread4 读到 1 个 99 和 2 个 100，这是因为 Thread3 在修改了数组的第一个变量后就被切换下来了，导致结果不符合预期。

加了读写锁的结果：

```
one@ONE-Y400:~/nachos/nachos-3.4/code/threads$ ./nachos
*** Thread 1 Name Read1 Read Shared Array 0: 100
Ticks: 200, TimeInterrept!
*** Thread 2 Name Read2 Read Shared Array 0: 100
*** Thread 2 Name Read2 Read Shared Array 1: 100
*** Thread 2 Name Read2 Read Shared Array 2: 100
Ticks: 400, TimeInterrept!
*** Thread 1 Name Read1 Read Shared Array 1: 100
*** Thread 1 Name Read1 Read Shared Array 2: 100
Ticks: 600, TimeInterrept!
*** Thread 3 Name Write1 Write Shared Array 0: 99
*** Thread 3 Name Write1 Write Shared Array 1: 99
Ticks: 800, TimeInterrept!
*** Thread 3 Name Write1 Write Shared Array 2: 99
*** Thread 4 Name Read3 Read Shared Array 0: 99
Ticks: 1000, TimeInterrept!
*** Thread 4 Name Read3 Read Shared Array 1: 99
*** Thread 4 Name Read3 Read Shared Array 2: 99
*** Thread 5 Name Write2 Write Shared Array 0: 98
Ticks: 1200, TimeInterrept!
*** Thread 5 Name Write2 Write Shared Array 1: 98
*** Thread 5 Name Write2 Write Shared Array 2: 98
Ticks: 1400, TimeInterrept!
*** Thread 6 Name Read4 Read Shared Array 0: 98
*** Thread 6 Name Read4 Read Shared Array 1: 98
*** Thread 6 Name Read4 Read Shared Array 2: 98
Ticks: 1600, TimeInterrept!
*** Thread 7 Name Write3 Write Shared Array 0: 97
*** Thread 7 Name Write3 Write Shared Array 1: 97
Ticks: 1800, TimeInterrept!
*** Thread 7 Name Write3 Write Shared Array 2: 97
*** Thread 8 Name Write4 Write Shared Array 0: 96
Ticks: 2000, TimeInterrept!
*** Thread 8 Name Write4 Write Shared Array 1: 96
*** Thread 8 Name Write4 Write Shared Array 2: 96
Ticks: 2200, TimeInterrept!
*** Thread 9 Name Read5 Read Shared Array 0: 96
*** Thread 9 Name Read5 Read Shared Array 1: 96
*** Thread 9 Name Read5 Read Shared Array 2: 96
Ticks: 2400, TimeInterrept!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 160, system 2440, 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...
```

这样读和写的结果都符合了预期，并且 Thread1 没有读完时 Thread2 仍然可以进行读，但在 Thread7 还没写完时，即使来了时钟中断，仍然会进行 7 的写操作，也符合我们对读写的要求。

第二部分

Exercise1

……

Exercise2

……

Exercise3

……

第三部分

Exercise1

……

第四部分

Exercise1

……

Exercise2

……

Exercise3

……

Exercise4

……

Exercise5

……

Exercise6

……

【对于阅读代码类的 exercise，请对其中你认为重要的部分（比如某文件，或某个类、或某个变量、或某个函数……）做出说明。

对于要编程实现的 exercise，请对你增加或修改的内容作出说明。如果增加或修改了某个类，请写出这个类写在那个文件中，它的的功能是什么，你自己添加或修改的成员变量

以及成员函数有哪些，它们的功能是什么；特别的，对于复杂的函数，请说明你的实现方法。不需要贴具体的实现代码。

要求：表述清楚即可， 可采用图表来辅助说明，不需要贴代码】

内容三：遇到的困难以及解决方法

困难 1

……

困难 2

……

【描述你在实习过程中遇到的困难，是与实习直接相关的技术方面的难题。突出说明你是如何攻克这些难关的。

要求：只需写一下有较深体会的困难。如果觉得整个过程都比较简单的话此部分可不用写。】

内容四：收获及感想

……

【自己的收获，任何关于实习的感想，可以是技术方面的或非技术方面的，可随意发挥。

要求：内容不限，体裁不限，字数不限，多多益善，有感而发。】

内容五：对课程的意见和建议

……

【请写下你认为课程需要改进的地方，任何方面，比如进度安排、难易程度、课堂讲解、

考核方式、题目设置……甚至如果你认为源代码哪里写得不好也欢迎提出。

各位同学反馈的信息对课程建设会有极大帮助。】

内容六：参考文献

操作系统 A 课件

【我们希望大家不要使用复制粘贴来拼凑你的报告。详细地列出你在完成 lab 的过程中引用的书籍，网站，讲义，包括你咨询过的大牛们。

要求：诚实，格式尽量按照论文的要求，请参考"论文参考文献格式.doc"】