lab3 同步机制

李糖 2001210320

任务完成情况

Exercises	Y/N
Exercise1	Y
Exercise2	Y
Exercise3	Y
Exercise4	Y
*challenge1	Y
*challenge2	Y

Exercise1 调研

> 调研Linux中实现的同步机制。

- Locking in the Linux Kernel

在`include/linux`路径下：

互斥锁

- `mutex.h`

其他锁

- `spinlock.h`
- `rwlock.h`
- ...

结论

Linux在内核中实现了很多种类不同的锁，通常情况下用于系统软件和硬件的管理。而对于用户级进程，据我所知一般是使用ptherad库。

Exercise2 源代码阅读

> 阅读下列源代码，理解Nachos现有的同步机制。
>
> code/threads/synch.h和code/threads/synch.cc
>
> code/threads/synchlist.h和code/threads/synchlist.cc

code/threads/synch.h(cc)

实现了信号量机制。

成员变量/函数	描述
int value	信号量值，永远大于等于0
List *queue	在P()中被阻塞的线程队列，等待value>0之后被唤醒
void P()	当value == 0时，将currentThread放入queue中。挂起currentThread并执行其他线程；当value>0时，value--
void V()	判断queue中是否有元素，如果有，则唤醒，并将其加入就绪队列；value++

code/threads/synchlist.h(cc)

基于List类和信号量机制，实现了一个同步链表。本次实验不会用到，这里不再赘述。

Exercise3 实现锁和条件变量

> 可以使用sleep和wakeup两个原语操作（注意屏蔽系统中断），也可以使用Semaphore作为唯一同步原语（不必自己编写开关中断的代码）。

在开头关中断，在结尾开中断，保证整个程序是原子操作。

Pthreads库

> pthreads提供了两种同步机制：mutex和condition

- POSIX Threads Programming
    - [Mutex Variables]
    - [Condition Variables]

Lock

Nachos已经有了一个Lock模板，我用semaphore来实现它。

我添加了两个private变量：

```
class Lock {
  ...
  private:
    ...
  Thread *heldThread;   //lab3 在isHeldByCurrentThread()使用
  Semaphore *semaphore; //信号量,在构造函数中将value初始化为1
};
```

当currentThread获得Lock的时候将heldThread指定为currentThread：

```
void Lock::Acquire()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);//关中断
    semaphore->P();
    heldThread = currentThread;
    DEBUG('l', "%s has aquired %s", heldThread->getName(), name); //l means lock
    interrupt->SetLevel(oldLevel);
}
```

当且仅当锁的拥有者为currentTHread才可以释放锁。

更多的细节请查看`code/thread/synch.cc`

Condition

Nachos已经给了Condition的模板，我用Lock来实现它。

我添加了一个private成员变量queue作为阻塞队列。

```
class Condition {
  private:
    List* queue; // 因某条件被阻塞的线程
};
```

注意到所有的Condition成员函数都需要一个参数conditionlock，这是因为使用条件变量的线程必须在之前就已经获得了锁。

用sigenal来唤醒单个线程，broadcast来唤醒多个线程：

```
void Condition::Signal(Lock *conditionLock)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //环境变量的所有者必须为当前线程
    ASSERT(conditionLock->isHeldByCurrentThread());
    //唤醒进程
    if (!queue->IsEmpty())
    {
        Thread *thread = (Thread *)queue->Remove();
        scheduler->ReadyToRun(thread);
        DEBUG('c', "%s wakes up \"%s\".\n", getName(), thread->getName());
    }
    interrupt->SetLevel(oldLevel);
}
```

```
void Condition::Broadcast(Lock *conditionLock)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //环境变量的所有者必须为当前线程
    ASSERT(conditionLock->isHeldByCurrentThread());
    DEBUG('c', "broadcast : ");
    //唤醒所有进程
    while (!queue->IsEmpty())
    {
        Thread *thread = (Thread *)queue->Remove();
        scheduler->ReadyToRun(thread);
        DEBUG('c', "%s\t", thread->getName());
```

```
    }
    DEBUG('c', "\n");
    interrupt->SetLevel(oldLevel);
}
```

关于wait()的实现请查看`code/thread/synch.cc`

测试

我将在exercise中进行Lock和Condition的测试。

Exercise4 生产者消费者

> 基于Nachos中的信号量、锁和条件变量，采用两种方式实现同步和互斥机制应用（其中使用条件变量实现同步互斥机制为必选题目）。具体可选择“生产者-消费者问题”、“读者-写者问题”、“哲学家就餐问题”、“睡眠理发师问题”等。（也可选择其他经典的同步互斥问题）

这里我选择实现生产者消费者问题，并使用Lock和Condition实现。代码框架可以参考：

> [生产者消费者--wiki百科]()

实现

```
//----------------------------------------------------------------------
// Lab3 Exercise4 生产者消费者问题
// 在main中new一个生产者和一个消费者
// 消费者：每次从buffer中取一个元素
// 生产者：每次生产一个元素放入buffer
// buff满,生产者阻塞，buff空，消费者阻塞
// 保证生产者和消费者互斥访问buffer
//----------------------------------------------------------------------
#define BUFFER_SIZE 5                   //buffer的大小
#define THREADNUM_P (Random() % 4 + 1) //生产者数,不超过4
#define THREADNUM_C (Random() % 4 + 1) //消费者数,不超过4
#define TESTTIME 500                    //本次测试的总时间

vector<char> buffer;      //方便起见，用STL作为buffer
Lock *mutex;              //mutex->缓冲区的互斥访问
Condition *full, *empty; //full->生产者的条件变量，empty->消费者的条件变量

//消费者线程
void Comsumer(int dummy)
{
    while (stats->totalTicks < TESTTIME) //约等于while(true),这样写可以在有限的时间内
结束
    {
        //保证对缓冲区的互斥访问
        mutex->Acquire();
        //缓冲区空，阻塞当前消费者
        while (!buffer.size())
        {
            printf("Thread \"%s\": Buffer is empty with size %d.\n",
currentThread->getName(), buffer.size());
            empty->Wait(mutex);
        }

        //消费一个缓冲区物品
```

```
        ASSERT(buffer.size());
        buffer.pop_back();
        printf("Thread \"%s\" gets an item.\n", currentThread->getName());

        //若存在阻塞的生产者，将他们中的一个释放
        if (buffer.size() == BUFFER_SIZE - 1)
            full->Signal(mutex);

        //释放锁
        mutex->Release();
        interrupt->OneTick(); //系统时间自增
    }
}

//生产者线程
void Producer(int dummy)
{
    while (stats->totalTicks < TESTTIME) //约等于while(true),这样写可以在有限的时间内
结束
    {
        //保证对缓冲区的互斥访问
        mutex->Acquire();

        //缓冲区满，阻塞当前线程,一定要使用while,如果用if，可能存在
        //这样一种情况:生产者1判断buffer满，阻塞；当它再次上处理机时，
        //buffer还是满的，但是它不会再判断了，而是直接进入了临界区
        while (buffer.size() == BUFFER_SIZE)
        {
            printf("Thread \"%s\": Buffer is full with size %d.\n",
currentThread->getName(), buffer.size());
            full->Wait(mutex);
        }

        //生产一个物品放入缓冲区
        ASSERT(buffer.size() < BUFFER_SIZE);
        buffer.push_back('0');
        printf("Thread \"%s\" puts an item.\n", currentThread->getName());

        //若存在阻塞的消费者，将他们中的一个释放
        if (buffer.size() == 1)
            empty->Signal(mutex);

        //释放锁
        mutex->Release();
        interrupt->OneTick(); //系统时间自增
    }
}
void Lab3ProducerAndComsumer()
{
    printf("Random created %d comsumers, %d producers.\n", THREADNUM_C,
THREADNUM_P);

    full = new Condition("Full_condition");   //初始化full
    empty = new Condition("Empty_condition"); //初始化empty
    mutex = new Lock("buffer_mutex");         //初始化mutex

    Thread *threadComsumer[THREADNUM_C];
    Thread *threadProducer[THREADNUM_P];
```

```
    //初始化消费者
    for (int i = 0; i < THREADNUM_C; ++i)
    {
        char threadName[20];
        sprintf(threadName, "Comsumer %d", i); //给线程命名
        threadComsumer[i] = new Thread(strdup(threadName));
        threadComsumer[i]->Fork(Comsumer, 0);
    }
    //初始化生产者
    for (int i = 0; i < THREADNUM_P; ++i)
    {
        char threadName[20];
        sprintf(threadName, "Producer %d", i); //给线程命名
        threadProducer[i] = new Thread(strdup(threadName));
        threadProducer[i]->Fork(Producer, 0);
    }
    // scheduler->Print();
    while (!scheduler->isEmpty())
        currentThread->Yield(); //跳过main的执行

    //结束
    printf("Producer consumer test Finished.\n");
}
```

测试

本次试验采用随机时间片模拟真实场景，在terminal中输入`./nachos -d c -rs -q 6`可查看结果：

> -d c means condition debug, -rs means random seed

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -d c -rs -q 6
Random created 2 comsumers, 3 producers.
Thread "Comsumer 0": Buffer is empty with size 0.
Empty_condition has blocked thread "Comsumer 0".
Thread "Comsumer 1": Buffer is empty with size 0.
Empty_condition has blocked thread "Comsumer 1".
Thread "Producer 0" puts an item.
Empty_condition wakes up "Comsumer 0".
Thread "Producer 0" puts an item.
======Random context switch, Ticks = 190=====
Thread "Producer 1" puts an item.
Thread "Producer 1" puts an item.
======Random context switch, Ticks = 250=====
Thread "Producer 2" puts an item.
Thread "Producer 2": Buffer is full with size 5.
Full_condition has blocked thread "Producer 2".
Thread "Producer 3": Buffer is full with size 5.
Full_condition has blocked thread "Producer 3".
Thread "Comsumer 0" gets an item.
Full_condition wakes up "Producer 2".
Thread "Comsumer 0" gets an item.
Thread "Comsumer 0" gets an item.
======Random context switch, Ticks = 420=====
Thread "Producer 0" puts an item.
Thread "Producer 0" puts an item.
Thread "Producer 0" puts an item.
```

```
======Random context switch, Ticks = 550=====
Thread "Producer 2": Buffer is full with size 5.
Full_condition has blocked thread "Producer 2".
Producer consumer test Finished.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 691, idle 101, system 590, 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...
```

结论

结果显示，成功使用semaphore和condition解决生产者消费者问题。

*challenge1 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。

背景知识

- [Wiki - Barrier (computer science)]
- [Latches And Barriers]

我仿造了`std::barrier`的`arrive_and_wait`的实现，并在`code/thread/synch.h`中添加了Barrier类：

```
class Barrier
{
public:
  Barrier(char *debugName, int num); // 构造函数
  ~Barrier();                        // 析构函数
  char *getName() { return (name); } // 调试用

  void stopAndwait(); // 在所有线程到达之前阻塞当前线程

private:
  char *name;             // 调试用
  int remain;             // 还剩多少线程没到?
  int threadNum;          // 线程总数
  Lock *mutex;            // condition中使用的锁
  Condition *condition; // 用来阻塞线程并唤醒他们
};
```

具体的实现请查看`code/thread/synch.cc`

在`code/thread/threadtest.cc`中编写了`Lab3Barrier()`函数，`testnum = 5`:

```
//----------------------------------------------------------------------
// lab3 Challenge1 Barrier
```

```
// new 4 个线程，每个线程分别对4个全局变量进行赋值
// 共分三个阶段,每个阶段赋值不同，但是在相同的阶段中，
// 每个线程对对应的数组元素赋值是相同的
//-----------------------------------------------------------------------

#define THREADNUM 4 //线程数
#define PHASENUM 3  //测试的阶段数
int num[THREADNUM];
Barrier *barrier;

//为每个变量赋值，变量与线程一一对应
void AssignValue(int i) //i代表数组线标
{
    //每个循环代表一个阶段
    for (int j = 1; j <= PHASENUM; ++j)
    {
        num[i] = j;
        printf("Phase %d: thread \"%s\" finished assignment, num[%d] = %d.\n", j,
currentThread->getName(), i, j);
        //多次增加时间片，使线程切换更频繁
        for (int i = 0; i < 4; ++i)
            interrupt->OneTick();
        barrier->stopAndWait(); //线程暂时被barrier阻塞，并等待所有线程抵达
    }
}

void Lab3Barrier()
{
    barrier = new Barrier("barrier", THREADNUM);
    Thread *threads[THREADNUM];
    //初始化线程和数组,并加入就绪队列
    for (int i = 0; i < THREADNUM; ++i)
    {
        num[i] = 0;
        char threadName[30];
        sprintf(threadName, "Barrier test %d", i); //给线程命名
        threads[i] = new Thread(strdup(threadName));
        threads[i]->Fork(AssignValue, i);
    }
    while (!scheduler->isEmpty())
        currentThread->Yield(); //跳过main的执行
    //结束
    printf("Barrier test Finished.\n");
}
```

预期结果：每个阶段中，数组num中的每个元素相等。

测试

在`terminal`中输入`./nachos -d b -q 5`可查看结果：

> -d b means barrier debug

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -d b -q 5
Phase 1: thread "Barrier test 0" finished assignment, num[0] = 1.
Thread "Barrier test 0" reached barrier with remain = 3.
Phase 1: thread "Barrier test 1" finished assignment, num[1] = 1.
```

```
Thread "Barrier test 1" reached barrier with remain = 2.
Phase 1: thread "Barrier test 2" finished assignment, num[2] = 1.
Thread "Barrier test 2" reached barrier with remain = 1.
Phase 1: thread "Barrier test 3" finished assignment, num[3] = 1.
Thread "Barrier test 3" reached barrier with remain = 0.
All threads reached barrier.
Phase 2: thread "Barrier test 3" finished assignment, num[3] = 2.
Thread "Barrier test 3" reached barrier with remain = 3.
Phase 2: thread "Barrier test 0" finished assignment, num[0] = 2.
Thread "Barrier test 0" reached barrier with remain = 2.
Phase 2: thread "Barrier test 1" finished assignment, num[1] = 2.
Thread "Barrier test 1" reached barrier with remain = 1.
Phase 2: thread "Barrier test 2" finished assignment, num[2] = 2.
Thread "Barrier test 2" reached barrier with remain = 0.
All threads reached barrier.
Phase 3: thread "Barrier test 2" finished assignment, num[2] = 3.
Thread "Barrier test 2" reached barrier with remain = 3.
Phase 3: thread "Barrier test 3" finished assignment, num[3] = 3.
Thread "Barrier test 3" reached barrier with remain = 2.
Phase 3: thread "Barrier test 0" finished assignment, num[0] = 3.
Thread "Barrier test 0" reached barrier with remain = 1.
Phase 3: thread "Barrier test 1" finished assignment, num[1] = 3.
Thread "Barrier test 1" reached barrier with remain = 0.
All threads reached barrier.
Barrier test Finished.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 320, idle 0, system 320, 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...
```

结果显示，共进行了三个阶段的赋值，每个阶段中，每个线程正确地对其负责的变量进行了赋值；在不同的阶段中，每个线程的赋值不同，符合预期，实验成功。为了测试在随机时间片下程序是否具有正确性，输入-rs 查看随机时间片下的结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ make;./nachos -d c -rs
-q 5
make: `nachos' is up to date.
Phase 1: thread "Barrier test 0" finished assignment, num[0] = 1.
Barrier condition has blocked thread "Barrier test 0".
Phase 1: thread "Barrier test 1" finished assignment, num[1] = 1.
Barrier condition has blocked thread "Barrier test 1".
Phase 1: thread "Barrier test 2" finished assignment, num[2] = 1.
===============Random context switch, Ticks = 190===============
Phase 1: thread "Barrier test 3" finished assignment, num[3] = 1.
Barrier condition has blocked thread "Barrier test 3".
broadcast : Barrier test 0      Barrier test 1  Barrier test 3
===============Random context switch, Ticks = 280===============
Phase 2: thread "Barrier test 2" finished assignment, num[2] = 2.
Barrier condition has blocked thread "Barrier test 2".
Phase 2: thread "Barrier test 0" finished assignment, num[0] = 2.
```

```
Barrier condition has blocked thread "Barrier test 0".
Phase 2: thread "Barrier test 1" finished assignment, num[1] = 2.
Barrier condition has blocked thread "Barrier test 1".
===============Random context switch, Ticks = 460===============
Phase 2: thread "Barrier test 3" finished assignment, num[3] = 2.
broadcast : Barrier test 2      Barrier test 0  Barrier test 1
Phase 3: thread "Barrier test 3" finished assignment, num[3] = 3.
===============Random context switch, Ticks = 580===============
Phase 3: thread "Barrier test 2" finished assignment, num[2] = 3.
Barrier condition has blocked thread "Barrier test 2".
Barrier condition has blocked thread "Barrier test 3".
Phase 3: thread "Barrier test 0" finished assignment, num[0] = 3.
Barrier condition has blocked thread "Barrier test 0".
Phase 3: thread "Barrier test 1" finished assignment, num[1] = 3.
broadcast : Barrier test 2      Barrier test 3  Barrier test 0
===============Random context switch, Ticks = 780===============
Barrier test Finished.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 916, idle 56, system 860, 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...
```

结论

在随机时间片下结果依然正确，我们的Barrier实现成功！

*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。

读者写者问题分为两类：

1. 读者优先（最简单）：任何读者都不能仅仅因为作家正在等待而等待其他读者完成。
2. 写者优先：写者准备就绪后，应该尽快让写者进入临界区。

两类问题都可能导致饥饿，其中第一类会导致写者饥饿，第二类会导致读者饥饿。

本次实验将实现第二类读写锁。

> Readers–writer lock - WIKIPEDIA

使用Condition和Lock

可以用一个条件变量，*COND*，一个普通的（互斥）锁，g，和各种计数器和标志描述当前处于活动状态或等待的线程。对于写优先的RW锁，可以使用两个整数计数器和一个布尔标志：

- *num_readers_active*：已获取锁的读者的数量（整数）
- *num_writers_waiting*：等待访问的写者数（整数）
- *writer_active*：写者是否已获得锁（布尔值）

最初*num_readers_active*和*num_writers_waiting*为零，而*writer_active*为false。

我将读写锁封装成一个类`code/threads/synch.h`：

```
class RWLock
{
public:
  RWLock(char *debugName);           // 构造函数
  ~RWLock();                         // 析构函数
  char *getName() { return (name); } // debug辅助

  // 读者锁
  void ReaderAcquire();
  void ReaderRelease();
  // 写者锁
  void WriterAcquire();
  void WriterRelease();

private:
  char *name;              // debug用
  int num_readers_active;  //已获取锁的读者的数量
  int num_writers_waiting; //等待访问的写者数（整数）
  int writer_active;       //写者是否已获得锁（布尔值）
  Condition *COND;         //条件变量COND
  Lock *g;                 //互斥锁g
};
```

读者锁

```
//Begin Read
void RWLock::ReaderAcquire()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //lock g
    g->Acquire();
    //writer first
    while (num_writers_waiting > 0 || writer_active)
        COND->Wait(g);
    // increamnet number of readers
    num_readers_active++;
    //unlock g
    g->Release();
    interrupt->SetLevel(oldLevel);
}

//End Read
void RWLock::ReaderRelease()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //lock g
    g->Acquire();
    //decrement number of readers
    num_reader_active--;
    //no readers active, notify COND
    if (!num_readers_active)
        COND->Broadcast(g);
    //unlock g
    g->Release();
```

```
    interrupt->SetLevel(oldLevel);
}
```

写者锁

```
//Begin Write
void RWLock::WriterAcquire()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //lock g
    g->Acquire();
    // increamnet number of writers
    num_writers_waiting++;
    //writer first
    while (num_readers_active > 0 || writer_active)
        COND->Wait(g);
    // decreamnet snumber of writers
    num_writers_waiting--;
    //set writer_active to true
    writer_active = true;
    //unlock g
    g->Release();
    interrupt->SetLevel(oldLevel);
}

//End Write
void RWLock::WriterRelease()
{
     IntStatus oldLevel = interrupt->SetLevel(IntOff);
    //lock g
    g->Acquire();
    //set writer_active to false
    writer_active = false;
    //notify COND
    COND->Broadcast(g);
    //unlock g
    g->Release();
    interrupt->SetLevel(oldLevel);
}
```

其他琐碎的代码请查看`code/threads/synch.cc`

测试

在`code/threads/threatest.cc`中编写了Lab3RWLock()函数，testnum = 7:

```
//----------------------------------------------------------------------
// lab3 Challenge2 RWLock
// 随机产生一定数量的读者和写者对临界资源buffer
// 进行读写；  写者的任务：写一句拿破仑的名言：
// "Victory belongs to the most persevering"
// 构造两个写者，第一个负责写前半部分，第二个负责写
// 后半部分，每个写者写一个字符就切换，观察是否会被
// 其他读者或者写者抢占使用权，如果不会，证明写者优
// 先实现成功
//----------------------------------------------------------------------
```

```
#define THREADNUM_R (Random() % 4 + 1)                          //读者数,不超过4
#define THREADNUM_W 2                                           //写者数
const string QUOTE = "Victory belongs to the most persevering."; //拿破仑的名言
const int QUOTE_SIZE = QUOTE.size();                            //长度
int shared_i = 0;                                               //写者公用，用于定
位，初始化为零
RWLock *rwlock;                                                 //读写锁
string RWBuffer;                                                //buffer

//写者线程
void Writer(int writeSize)
{
    while (shared_i < writeSize)
    {
        rwlock->WriterAcquire();
        RWBuffer.push_back(QUOTE[shared_i++]);
        printf("%s is writing: %s\n", currentThread->getName(),
RWBuffer.c_str());
        //让每个写者写一个字符就切换一次，看会不会被其他的读者或者写者抢占。
        currentThread->Yield();
        rwlock->WriterRelease();
    }
}

//读者线程
void Reader(int dummy)
{
    while (shared_i < QUOTE_SIZE)
    {
        rwlock->ReaderAcquire();
        printf("%s is reading :%s\n", currentThread->getName(),
RWBuffer.c_str());
        rwlock->ReaderRelease();
    }
}

void Lab3RWLock()
{
    printf("Random created %d readers, %d writers.\n", THREADNUM_R, THREADNUM_W);

    rwlock = new RWLock("RWLock"); //初始化rwlock
    Thread *threadReader[THREADNUM_R];
    Thread *threadWriter[THREADNUM_W];

    //初始化写者
    for (int i = 0; i < THREADNUM_W; ++i)
    {
        char threadName[20];
        sprintf(threadName, "Writer %d", i); //给线程命名
        threadWriter[i] = new Thread(strdup(threadName));
        int val = !i ? QUOTE_SIZE - 20 : QUOTE_SIZE;
        threadWriter[i]->Fork(Writer, val);
    }
    //初始化读者
    for (int i = 0; i < THREADNUM_R; ++i)
    {
        char threadName[20];
        sprintf(threadName, "Reader %d", i); //给线程命名
```

```
        threadReader[i] = new Thread(strdup(threadName));
        threadReader[i]->Fork(Reader, 0);
    }

    while (!scheduler->isEmpty())
        currentThread->Yield(); //跳过main的执行

    //结束
    printf("Secondary Reader Writer test Finished.\n");
}
```

在terminal中输入`./nachos -q 7`可查看结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ make;./nachos -q 7
make: `nachos' is up to date.
Random created 2 readers, 2 writers.
Writer 0 is writing: V
Writer 0 is writing: Vi
Writer 0 is writing: Vic
Writer 0 is writing: Vict
Writer 0 is writing: Victo
Writer 0 is writing: Victor
Writer 0 is writing: Victory
Writer 0 is writing: Victory
Writer 0 is writing: Victory b
Writer 0 is writing: Victory be
Writer 0 is writing: Victory bel
Writer 0 is writing: Victory belo
Writer 0 is writing: Victory belon
Writer 0 is writing: Victory belong
Writer 0 is writing: Victory belongs
Writer 0 is writing: Victory belongs
Writer 0 is writing: Victory belongs t
Writer 0 is writing: Victory belongs to
Writer 0 is writing: Victory belongs to
Writer 0 is writing: Victory belongs to t
Writer 1 is writing: Victory belongs to th
Writer 1 is writing: Victory belongs to the
Writer 1 is writing: Victory belongs to the
Writer 1 is writing: Victory belongs to the m
Writer 1 is writing: Victory belongs to the mo
Writer 1 is writing: Victory belongs to the mos
Writer 1 is writing: Victory belongs to the most
Writer 1 is writing: Victory belongs to the most
Writer 1 is writing: Victory belongs to the most p
Writer 1 is writing: Victory belongs to the most pe
Writer 1 is writing: Victory belongs to the most per
Writer 1 is writing: Victory belongs to the most pers
Writer 1 is writing: Victory belongs to the most perse
Writer 1 is writing: Victory belongs to the most persev
Writer 1 is writing: Victory belongs to the most perseve
Writer 1 is writing: Victory belongs to the most persever
Writer 1 is writing: Victory belongs to the most perseveri
Writer 1 is writing: Victory belongs to the most perseverin
Writer 1 is writing: Victory belongs to the most persevering
Writer 1 is writing: Victory belongs to the most persevering.
Reader 0 is reading :Victory belongs to the most persevering.
```

```
Reader 1 is reading :Victory belongs to the most persevering.
Secondary Reader Writer test Finished.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1760, idle 0, system 1760, 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...
```

输入`./nachos -d c -q 7`可以查看线程阻塞信息：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ make;./nachos -d c -q
7
make: `nachos' is up to date.
Random created 2 readers, 2 writers.
Writer 0 is writing: V
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Vi
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Vic
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Vict
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victo
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victor
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory
Condition has blocked thread "Writer 1".
```

```
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory b
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory be
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory bel
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belo
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belon
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belong
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belongs
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belongs
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belongs t
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belongs to
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 0 is writing: Victory belongs to
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
```

```
Writer 0 is writing: Victory belongs to t
Condition has blocked thread "Writer 1".
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Writer 1    Reader 0        Reader 1
Writer 1 is writing: Victory belongs to th
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the m
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the mo
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the mos
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most p
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most pe
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most per
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most pers
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most perse
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most persev
```

```
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most perseve
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most persever
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most perseveri
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most perseverin
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most persevering
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Writer 1 is writing: Victory belongs to the most persevering.
Condition has blocked thread "Reader 0".
Condition has blocked thread "Reader 1".
broadcast : Reader 0    Reader 1
Reader 0 is reading :Victory belongs to the most persevering.
broadcast :
Reader 1 is reading :Victory belongs to the most persevering.
broadcast :
Secondary Reader Writer test Finished.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1760, idle 0, system 1760, 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...
```

结论

结果显示，在写者写完之前，读者和其它写者会被一直阻塞，直到写者完成自己的任务之后，其它写者才能进行写，没有写者写了，读者才能进行读，并且多个读者之间可以并发读，这表明我们的RWLock实现成功。

challenge 3 研究Linux的kfifo机制是否可以移植到Nachos上作为一个新的同步模块

todo