# Lab3 线程同步 实验报告

## 内容一：总体概述

在了解了linux同步机制后，在nachos中实现了锁与条件变量并在生产者-消费者、读写者问题上应用。

## 内容二：任务完成情况

### 任务完成列表

| Exercise1 | Exercise2 | Exercise3 | Exercise4 | Challenge2 |
|---|---|---|---|---|
| Y | Y | Y | Y | Y |

### Exercise 1 调研

> 调研Linux中实现的同步机制

#### 原子操作

Linux 中最简单的同步方法就是原子操作。由于 C 不能实现原子操作，因此 Linux 依靠底层架构来提供这项功能。各种底层架构存在很大差异，因此原子函数的实现方法也各不相同。一些方法完全通过汇编语言来实现，而另一些方法依靠 c 语言并且使用 `local_irq_save` 和 `local_irq_restore` 禁用中断。

#### 自旋锁

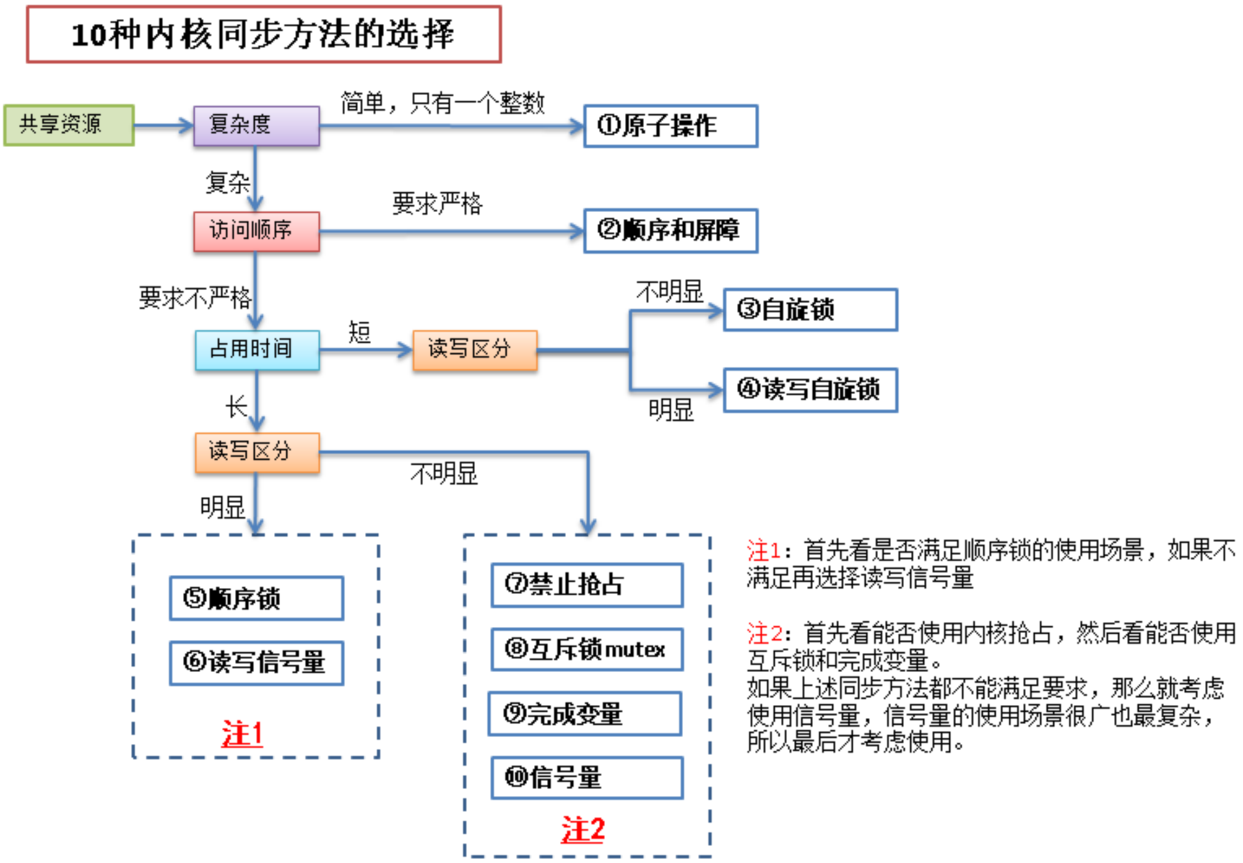
一般只用于多cpu系统。自旋锁的特点就是当一个线程获取了锁之后，其他试图获取这个锁的线程一直在循环等待获取这个锁，直至锁重新可用。由于线程一直在循环获取这个锁，所以会造成 CPU 处理时间的浪费，因此最好将自旋锁用于很快能处理完的临界区。

#### 信号量

信号量也是一种锁，和自旋锁不同的是，线程获取不到信号量的时候，不会像自旋锁一样循环区试图获取锁，**而是进入睡眠**，直至有信号量释放出来时，才会唤醒睡眠的线程，进入临界区执行。

#### 互斥锁(mutex)

与二元信号量类似，区别在于互斥锁只能由同一个进程获取和释放、持有时进程不能退出、不能递归上锁和解锁。




## Exercise 2 源代码阅读

> 阅读下列源代码，理解Nachos现有的同步机制。
>
> 1. code/threads/synch.h和code/threads/synch.cc
> 2. code/threads/synchlist.h和code/threads/synchlist.cc

### synch.h和synch.cc

`synch.h`和`synch.cc`中定义了Nachos用于线程同步的三种数据结构：信号量、锁与条件变量。其中信号量已经实现。

- 信号量：信号量中有PV操作、私有变量value表示信号量大小、queue为等待该信号量的线程的队列。

  P操作每次先关中断，检查资源是否为0，是则将自己放入该信号量的等待队列并使自己睡眠；否则使资源减一并开中断，继续运行。

```
void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts

    while (value == 0) {       // semaphore not available
  queue->Append((void *)currentThread); // so go to sleep
  currentThread->Sleep();
    }
    value--;            // semaphore available,
            // consume its value

    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts
}
```

V操作先关中断，检测信号量的等待队列中是否有线程，有则唤醒；将资源加一，开中断继续运行。

```
void Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)     // make thread ready, consuming the V
immediately
  scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

- 锁与条件变量：条件变量要与锁配合使用，维护一个等待队列。下面都是原子操作。

```
void Lock::Acquire() {} //获得锁，若被其他线程占用则sleep
void Lock::Release() {} //释放锁，若有线程在等待锁则唤醒他
void Condition::Wait(Lock* conditionLock) { ASSERT(FALSE); } //释放锁，睡眠并进入等待队列，被唤醒后取得锁
void Condition::Signal(Lock* conditionLock) { } //从等待队列唤醒一个线程
void Condition::Broadcast(Lock* conditionLock) { } //唤醒所有等待队列中的线程
```

## Exercise 3 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）。

简单阐述：条件变量只有一个等待队列，需要配合锁使用。锁可以用信号量实现。锁的获取释放、条件变量的睡眠唤醒、信号量的PV都实现为原子操作。

### 锁

这里使用信号量实现锁，在Lock类中增加一个信号量（初始化都为1）、一个指明锁的持有者的线程指针。

```
private:
    char* name;        // for debugging
    // plus some other stuff you'll need to define
    Semaphore *semaphore;
    Thread *holdingThread;
```

在`synch.cc`中实现构造函数、Acquire()、Release():

```
Lock::Lock(char* debugName) {
    name = debugName;
    semaphore = new Semaphore("Mutex",1);
}

//获得锁，若被其他进程占用则sleep
void Lock::Acquire() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    semaphore->P();
    holdingThread = currentThread;
    (void) interrupt->SetLevel(oldLevel);
}

//释放锁，若有进程在等待锁则唤醒他
void Lock::Release() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    semaphore->V();
    holdingThread = NULL;
    (void) interrupt->SetLevel(oldLevel);
}
```

### 条件变量

在Condition类中增加一个等待队列，存储阻塞在这个条件变量的线程

```
private:
    char* name;
    // plus some other stuff you'll need to define
    List * waitList;
```

在synch.cc中实现构造函数、Wait()、Signal()、Broadcast()

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

```
    waitList = new List;
}
Condition::~Condition() {
    delete waitList;
}

//释放锁，睡眠并进入等待队列，被唤醒后取得锁
void Condition::Wait(Lock* conditionLock) {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    conditionLock->Release();
    waitList->Append((Thread *)currentThread);

    printf("Thread %s sleeping\n", currentThread->getName());
    currentThread->Sleep();
    printf("Thread %s waking up\n", currentThread->getName());
    conditionLock->Acquire();
    (void) interrupt->SetLevel(oldLevel);
}

//从等待队列唤醒一个线程
void Condition::Signal(Lock* conditionLock) {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(conditionLock->isHeldByCurrentThread());
    if(conditionLock->isHeldByCurrentThread()){
        Thread *thread = (Thread *)waitList->Remove();
        if(thread){
            scheduler->ReadyToRun(thread);
            printf("Thread %s signaled %s\n", currentThread->getName(), thread-
>getName());
        }

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

//唤醒所有等待队列中的线程
void Condition::Broadcast(Lock* conditionLock) {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    while(!waitList->IsEmpty()){
        Signal(conditionLock);
    }
    (void) interrupt->SetLevel(oldLevel);
}
```

与exercise4一起测试

## Exercise 4 实现同步互斥实例

> 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）

下面锁和条件变量实现生产者-消费者问题（一个生产者，两个消费者，通过-rs随机时间中断切换线程）：

```
void L3E4_Con_condition(int which){
    for(int i = 0; i < 6; i ++){
        lock->Acquire();
        //要用while，而不是if，因为线程从wait中被唤醒后，再上cpu后应该重新判断资源情况
        while(product == 0){
            printf("There is no more product!\n");
            consumeCondition->Wait(lock);
        }
        product--;
        printf("Thread %s consumed a product, %d products now\n",
currentThread->getName(), product);
        produceCondition->Signal(lock); //只要消费一个就可以唤醒生产者了。
        lock->Release();
    }
}

void L3E4_Pro_condition(int which){
    for(int i = 0; i < 10; i ++){
        lock->Acquire();
        while(product == 3){
            printf("There are too much products!\n");
            produceCondition->Wait(lock);
        }
        product++;
        printf("Thread %s produced a product, %d products now\n",
currentThread->getName(), product);
        consumeCondition->Signal(lock); //只要生产一个就可以唤醒消费者了。
        lock->Release();
    }
}

void Lab3Exercise4()
{
    DEBUG('t', "Entering Lab3Challenge3");
    lock = new Lock("Pro_cons_Lock");
    produceCondition = new Condition("ProduceCondition");
    consumeCondition = new Condition("ConsumeCondition");

    Thread* consumer1= new Thread("Consumer1");
    consumer1->Fork(L3E4_Con_condition, consumer1->getTid());
```

```
    Thread* consumer2= new Thread("Consumer2");
    consumer2->Fork(L3E4_Con_condition, consumer2->getTid());

    Thread* producer= new Thread("Producer");
    producer->Fork(L3E4_Pro_condition, producer->getTid());

}
```

测试：命令行运行 ./nachos -rs -q 6：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -rs -q 6
There is no more product!
Thread Consumer2 sleeping
Thread Producer produced a product, 1 products now
Thread Producer signaled Consumer2
Thread Consumer2 waking up
Thread Producer produced a product, 2 products now
Thread Consumer1 consumed a product, 1 products now
Thread Consumer1 consumed a product, 0 products now
There is no more product!
Thread Consumer2 sleeping
Thread Producer produced a product, 1 products now
Thread Producer signaled Consumer2
Thread Consumer2 waking up
Thread Producer produced a product, 2 products now
Thread Producer produced a product, 3 products now
There are too much products!
Thread Producer sleeping
Thread Consumer2 consumed a product, 2 products now
Thread Consumer2 signaled Producer
Thread Producer waking up
Thread Consumer1 consumed a product, 1 products now
Thread Consumer1 consumed a product, 0 products now
There is no more product!
Thread Consumer2 sleeping
Thread Producer produced a product, 1 products now
Thread Producer signaled Consumer2
Thread Producer produced a product, 2 products now
Thread Consumer2 waking up
Thread Producer produced a product, 3 products now
There are too much products!
Thread Producer sleeping
Thread Consumer2 consumed a product, 2 products now
Thread Consumer2 signaled Producer
Thread Producer waking up
Thread Consumer2 consumed a product, 1 products now
Thread Consumer2 consumed a product, 0 products now
```

```
Thread Producer produced a product, 1 products now
Thread Producer produced a product, 2 products now
Thread Consumer2 consumed a product, 1 products now
Thread Consumer1 consumed a product, 0 products now
There is no more product!
Thread Consumer2 sleeping
There is no more product!
Thread Consumer1 sleeping
No threads ready or runnable, and no pending interrupts.
```

可见成功实现。

## Challenge 2 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。

与生产消费的区别：同时可以有多个读者

```
//---------------------------
//用锁实现读者写者问题
//命令行运行 ./nachos -rs -q 7
//---------------------------

void L3C2_read(int which){
    for(int i = 0; i < 6; i++){
        rclock->Acquire();
        readerCount++;
        if(readerCount == 1)
            rwlock->Acquire();
        rclock->Release();

        printf("%s is reading with %d readers\n", currentThread->getName(),
readerCount - 1);
        interrupt->OneTick(); //这里使时钟前进，制造多读者共存的机会
        rclock->Acquire();
        readerCount--;
        if(readerCount == 0)
            rwlock->Release();
        rclock->Release();
    }
}

void L3C2_write(int which){
    for(int i = 0; i < 10; i++){
        rwlock->Acquire();
        printf("%s is writing\n", currentThread->getName());
```

```
        rwlock->Release();
    }
}

void Lab3Challenge2()
{
    DEBUG('t', "Entering Lab3Challenge2");
    rclock = new Lock("ReaderCountLock");
    rwlock = new Lock("ReaderWriterLock");
    Thread* reader1= new Thread("Reader1");
    reader1->Fork(L3C2_read, reader1->getTid());

    Thread* reader2= new Thread("Reader2");
    reader2->Fork(L3C2_read, reader2->getTid());

    Thread* reader3= new Thread("Reader3");
    reader3->Fork(L3C2_read, reader3->getTid());

    Thread* writer= new Thread("Writer");
    writer->Fork(L3C2_write, writer->getTid());

}
```

如果使用之前的Lock的Release()，可能出现情况：第一个进来的读者和最后一个离开的读者不是同一个，这样就无法释放读写锁，权衡之下我取消了对锁的持有者的判断：

```
void Lock::Release() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // ASSERT(holdingThread == currentThread);
    semaphore->V();
    holdingThread = NULL;
    (void) interrupt->SetLevel(oldLevel);
}
```

测试：命令行运行 ./nachos -rs -q 7：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -rs -q 7
Reader1 is reading with 0 readers
Reader2 is reading with 1 readers
Reader3 is reading with 1 readers
Reader2 is reading with 0 readers
Writer is writing
Writer is writing
Writer is writing
Reader1 is reading with 1 readers
Reader3 is reading with 0 readers
Reader2 is reading with 1 readers
Reader1 is reading with 1 readers
```

```
Reader2 is reading with 2 readers
Reader2 is reading with 2 readers
Reader1 is reading with 2 readers
Reader3 is reading with 2 readers
Reader3 is reading with 2 readers
Reader2 is reading with 0 readers
Writer is writing
Writer is writing
Writer is writing
Writer is writing
Writer is writing
Writer is writing
Writer is writing
Reader3 is reading with 1 readers
Reader1 is reading with 1 readers
Reader3 is reading with 1 readers
Reader1 is reading with 0 readers
No threads ready or runnable, and no pending interrupts.
```

可见成功实现

# 内容三 遇到的困难及解决方法

- 读写者问题中，可能出现情况：第一个进来的读者和最后一个离开的读者不是同一个，这样就无法释放读写锁，权衡之下我取消了对锁的持有者的判断
- 在nachos中默认是运行在单cpu上，所以pv操作的原子性可以通过开关中断来保证，那多cpu中如何保证呢？

  关键在于寄存器与内存的**“交换”**可以是原子操作，而读取变量和修改变量不是原子操作。

  > 变量locked代表锁的状态，当locked为1表示当前访问的数据结构已经锁住，locked为0表示未锁住，能够访问当前数据结构，spinlock还附带有调试信息，比如锁的名字，当前占有锁的cpu和调用栈。
  >
  > 实际上按照普通方式访问并修改变量locked本身就存在竞争条件，可能有两个CPU同时读取locked的值为0，并认为都可以获得锁占用数据结构，然后将locked置1占用锁，实际上有两个CPU同时获得了锁，违反了访问该数据结构的互斥性，出现这种现象的根本原因是读取变量和修改变量不是一个原子操作，如果读取变量和修改变量这两个过程连续进行不可打断，并在访问时只允许一个CPU进行，便能实现访问locked的原子操作，xv6使用了x86架构下的xchg指令实现，xchg原子地交换一个寄存器和内存字的值，通过循环反复xchg，如果返回的内存字值为1，代表有CPU或者进程占用锁，继续循环等待不断xchg，如果xchg返回为0，代表目前没有人占用锁，通过将锁置1占用数据结构，然后跳出循环，通过这样的实现，每个CPU或进程在访问一个可能出现竞争的数据结构时，必须提前获得这个数据的锁，如果暂时无法获得锁则不断循环测试自旋，在处理好了数据后再解除锁的占用以便另一个CPU或者进程能重新占用锁。

# 内容四 收获及感想

熟悉了进程同步，对信号量、锁、条件变量有了清晰的了解。

# 内容五 对课程的意见或建议

无。

# 参考文献


[内核中各种同步机制]

[互斥锁与条件变量]

[xv6源码分析-锁]