同步机制实习报告

杨东升 1400012898
2017.3.27

目录

内容一：总体概述 3
内容二：任务完成情况 3
任务完成列表（Y/N） 3
具体 Exercise 的完成情况 3
内容三：遇到的困难以及解决方法 11
内容四：收获及感想 12
内容五：对课程的意见和建议 12
内容六：参考文献 12

内容一：总体概述

本实验重点研究线程同步的三种机制，分别是信号量，锁和条件变量。三种机制互为补充，具有区别的同时也具有密切的联系。在本实验中需要对线程同步的具体问题实例进行分析和解决。此外，本实验还需要实现读写锁和 Barrier，并给出测试样例。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Challenge1	Challenge2	Challenge3
第一部分	Y	Y	Y	Y	Y	Y	Y

具体 Exercise 的完成情况

Exercise1: 调研 Linux 中实现的同步机制

linux 下提供了多种方式来处理线程同步，最常用的是互斥锁、条件变量和信号量。

1) 通过锁机制实现线程间的同步:

锁是一个全局的标识符，每个锁在一个时刻只能被一个线程获得，只有获得锁的线程才能执行关于共享资源的操作，执行结束后再释放锁。相关操作包括：

初始化锁；

加锁。对共享资源的访问，要对互斥量进行加锁，如果互斥量已经上了锁，调用线程会阻塞，直到互斥量被解锁；

解锁。在完成了对共享资源的访问后，要对互斥量进行解锁；

销毁锁。锁在是使用完成后，需要进行销毁以释放资源。

2) 通过条件变量实现线程间的同步:

条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机

制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。相关操作包括：

初始化条件变量；

等待条件成立。释放锁，同时阻塞线程；

激活条件变量。使得它的一个等待线程能够运行；

清除条件变量，所有等待线程结束后方能正常清除。

3) 如同进程一样，线程也可以通过信号量来实现通信。信号量一般是一个整型变量，不同线程通过 P，V 操作可以改变它的值。通过检查信号量的值就知道是否可以访问共享资源。涉及的操作有以下四种：

信号量初始化；

等待信号量。给信号量减 1，然后等待直到信号量的值大于 0；

释放信号量。信号量值加 1。并通知其他等待线程；

销毁信号量。我们用完信号量后都它进行清理。归还占有的一切资源。

Exercise2: 源代码阅读

synch.h 中定义了三种同步机制，synch.cc 只实现了信号量机制。

Semaphore 类是信号量的具体实现。包含两个重要的私有变量，一个是 value，代表信号量的值，另一个是 queue，代表等待队列。P 和 V 是对外的接口，P 操作把线程挂起，加入等待队列。在恢复执行后把 value 减 1。V 操作把 value 加 1，把等待队列的第一个线程加入就绪队列。这两个操作都是原子操作，所以执行时要关中断。

Lock 类定义了互斥锁的方法。成员函数有 acquier 和 release，分别用来占有锁和释放锁。isHeldByCurrentThread 函数用来判断锁是否被占据。

Condition 类定义了条件变量的方法。wait 函数用来等待条件变量，signal 函数用来唤醒一个等待此条件变量的线程，broadcast 函数用来唤醒所有等待此条件变量的线程。

SynchList.h 和 SynchList.cc 对 List 类进行封装，实现了一个可以互斥访问的 list

包括增加元素，提取元素和对所有元素操作的函数。在增删和统一操作的时候都需要对 list 加锁。在删除的时候如果 list 为空则需要等待；在增加的时候发出 list 为空的激活信号。

Exercise3: 实现锁和条件变量

在 synch.h 中：

在 Lock 类中添加了如下变量：

Semaphore* owned;　　//利用信号量来实现锁

Thread* owner;　　//记录锁的持有者

在 Condition 类中添加了如下变量：

queue;　　//等待条件变量的队列

在 synch.cc 中：

实现了 Lock 类的函数：

构造函数对 owner 赋空值，初始化 owned 信号量

Acquire() 屏蔽中断，对 owned 使用 P 操作，如果成功返回则把 owner 改为 currentThread

Release() 屏蔽中断，对 owned 使用 V 操作，并把 owner 赋空值

isHeldByCurrentThread() 通过判断 owner 来返回锁是否被当前线程持有，作为 Acquire() 是否成功的判断依据。

实现了 Condition 类的函数：

构造函数初始化队列 queue

Wait() 屏蔽中断，释放参数中的锁，加入等待队列，下 CPU。上 CPU 后再获取锁。

Signal() 屏蔽中断，如果等待队列不空就把第一个线程加入就绪队列。

Broadcast() 屏蔽中断，循环地把等待队列中的所有线程加入就绪队列。

在 ThreadTest.cc 中：

编写 TestForLock() 函数测试锁机制。该函数创建两个 LockThread 线程，第一个会在执行一半时 Yield，第二个不会 Yield。进程调度采用时间片轮转机制。

LockThread() 函数的代码如下：

```
void LockThread(int arg)
{
    printf("Begin ") ;
    while( !lock->isHeldByCurrentThread() ) lock->Acquire() ;
    if(arg)
    {
        printf("Yield ") ;
        currentThread->Yield() ;
        printf("Continue ") ;
    }
    printf("Finish ") ;
    lock->Release() ;
}
```

运行结果如下：

从结果中可以看出在第一个线程 Yield 后，第二个线程得以开始运行，但是无法进入临界区，而是在忙等。直到第一个进程又被唤醒，并完成任务，释放锁，第二个线程才正常运行并完成任务。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 5

Switching from thread "main" to thread "thread 1"
Begin Yield
Switching from thread "thread 1" to thread "thread 2"
Begin
Switching from thread "thread 2" to thread "thread 1"
Continue Finish
Switching from thread "thread 1" to thread "thread 2"
Finish No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

在 ThreadTest.cc 中：

编写 TestForCond() 函数测试条件变量机制。该函数创建两个 CondThread 线程，第一个会在执行一半时 Wait，第二个会 Signal。进程调度采用时间片轮转机制。

CondThread() 函数首先获取锁，然后判断参数，如果参数是 1，则 Wait()，如果参数是 0，则 Signal()，然后释放锁。执行过程中会打印运行信息。

运行结果如下：

可以看出线程 1 等待并挂起，线程 2 正常运行没有死锁，并唤醒了线程 1，然后线程 2 和线程 1 先后结束运行。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$./nachos -q 6

Switching from thread "main" to thread "thread 1"
Begin
Wait

Switching from thread "thread 1" to thread "thread 2"
Begin
Signal
Finish

Switching from thread "thread 2" to thread "thread 1"
Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Exercise4: 实现生产者——消费者问题：

无论是用条件变量还是信号量处理生产者消费者问题，我都是在 ThreadTest.cc 中，实现 produce 和 consume 函数，然后再用 Produce 函数和 Consume 函数多次调用 produce 和 consume 来测试同步处理是否正确。

一. 使用锁和条件变量

此同步机制使用两个条件变量 full, empty 以及一个锁 lock
produce 函数首先获取锁，然后判断缓冲区是否满，如果满，则调用 full->wait()；接下来把数据放入缓冲区中，如果缓冲区之前为空，则调用 empty->Signal()，最后释放锁。
consume 函数首先获取锁，然后判断缓冲区是否空，如果空，则调用 empty->wait()；接下来从缓冲区中提取数据，如果缓冲区之前为满，则调用 full->Signal()，最后释放锁。
Produce 和 Consume 分别调用了 produce 和 consume 十次。

测试时，先初始化锁和条件变量，设置缓冲区大小为 3。再分别 fork 一个 Produce 线程，一个 Consume 线程。使用先来先服务线程调度机制。

运行结果如下图。可以看到每个线程在执行三次后，由于缓冲区满或者空，会挂起，另外一个线程会开始执行。具体实现时，一个线程唤醒另一个线程后，仍然会自己占据 CPU，直到自己放弃执行。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 7
No test specified.

Switching from thread "main" to thread "Producer"
produce, id = 1
produce, id = 2
produce, id = 3
producer waiting, num = 3

Switching from thread "Producer" to thread "Consumer"
consume, id = 3
consume, id = 2
consume, id = 1
consumer waiting, num = 0

Switching from thread "Consumer" to thread "Producer"
producer waking, num = 0
produce, id = 4
produce, id = 5
produce, id = 6
producer waiting, num = 3

Switching from thread "Producer" to thread "Consumer"
consumer waking, num = 3
consume, id = 6
consume, id = 5
consume, id = 4
consumer waiting, num = 0
```

二. 使用信号量

此同步机制使用一个锁，三个信号量 Full, Empty, Mutex
produce 函数首先对 Full 进行 P 操作，防止缓冲区满；接下来对 Mutex 进行 P 操作，保护临界区，并把数据放入缓冲区中，完成数据操作后对 Mutex 进行 V 操作，最后对 Empty 进行 V 操作，说明有缓冲区不空。
consume 函数首先对 Empty 进行 P 操作，防止缓冲区空；接下来利用 Mutex 的 P，V 操作保护临界区，并从缓冲区中提取数据，最后对 Full 进行 V 操作。
Produce 和 Consume 分别调用了 produce 和 consume 十次。

测试时，先初始化锁和条件变量，假设缓冲区大小为 3，我就设置 Full 的初值为 3，Empty 的初值为 0，Mutex 的初值为 1。再分别 fork 一个 Produce 线程，一个 Consume 线程。使用先来先服务线程调度机制。

运行结果如下图。可以看到每个线程在执行三次后，由于缓冲区满或者空，会挂起，另外一个线程会开始执行。具体实现时，一个线程唤醒另一个线程后，仍然会自己占据 CPU，直到自己放弃执行。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 7
No test specified.

Switching from thread "main" to thread "Producer"
produce, id = 1
produce, id = 2
produce, id = 3

Switching from thread "Producer" to thread "Consumer"
consume, id = 3
consume, id = 2
consume, id = 1

Switching from thread "Consumer" to thread "Producer"
produce, id = 4
produce, id = 5
produce, id = 6

Switching from thread "Producer" to thread "Consumer"
consume, id = 6
consume, id = 5
consume, id = 4
```

Challenge1: 实现 barrier：

我创建了一个 Barrier 类，包含一个私有的信号量 sem 和一个私有的条件变量 cond，前者用来统计多少个线程到达 Barrier，后者用于主线程唤醒所有线程。公有函数 Wait 用于等待所有线程到达同一点并发出 Broadcast。公有函数 Reach 用于表示某一个线程到达了 Barrier。

信号量 sem 的初始值设置为负的线程数加 1
主线程创建其他线程后，调用 Barrier->Wait()，它会对 sem 进行 P 操作来等待所有线程到达 Barrier，等待完成后会对 cond 进行 Broadcast 操作。
其他线程被创建后，到达截止点时，会调用 Barrier->Reach()，它会对 sem 进行 V 操作，并对 cond 进行 Wait 操作。
当所有线程都到达截止点，V 操作的数量正好能使主线程被唤醒。然后如上所述，主线程调用 cond->Broadcast，使所有线程可以接着运行。

实验效果如下图所示。可以看到所有 3 个线程执行完上半部分后，才接下来执行下半部分。使用先来先服务线程调度机制。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 8
waiting for every thread reaching the barrier

Switching from thread "main" to thread "Thread1"
Thread 1 doing the first part

Switching from thread "Thread1" to thread "Thread2"
Thread 2 doing the first part

Switching from thread "Thread2" to thread "Thread3"
Thread 3 doing the first part

Switching from thread "Thread3" to thread "main"
every thread reaching the barrier
No test specified.

Switching from thread "main" to thread "Thread1"
Thread 1 doing the second part

Switching from thread "Thread1" to thread "Thread2"
Thread 2 doing the second part

Switching from thread "Thread2" to thread "Thread3"
Thread 3 doing the second part
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Challenge2: 实现读写锁：

我实现了两个函数，myRead 和 myWrite，定义了全局变量 reader 表示运行的读者数量，定义了互斥信号量 Mutex，和缓冲区保护信号量 Buf。

在 myRead 函数中，首先对 Mutex 进行 P 操作，然后增加读者数量，如果读者数量为 1，则对 Buf 进行 P 操作，再对 Mutex 进行 V 操作。接下来读取缓冲区。之后为了测试多读者并行的情况，在这里进行 Yield。唤醒后，再对 Mutex 进行 P 操作，然后减少读者数量，如果读者数量为 0，则对 Buf 进行 V 操作，再对 Mutex 进行 V 操作。

在 myWrite 函数中，首先对 Buf 进行 P 操作，然后对缓冲区进行写操作，然后对 Buf 进行 V 操作。

在 TestForWR 函数中，初始化信号量和缓冲区，然后分别创建 4 个写线程和 4 个读线程。采用 random 时间片轮转调度算法。

运行结果如下图。可以看到写线程只能一个一个运行，每个结束后下个才能运行。读线程可以并发执行，在一个读线程结束前另一个读线程可以读出数据。我使用的是读者优先策略。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 9
writer 1 writes: a
writer 1 finishes
writer 2 writes: ab
writer 2 finishes
reader 2 reads: ab
reader 2 yields
reader 3 reads: ab
reader 3 yields
reader 4 reads: ab
reader 4 yields
reader 1 reads: ab
reader 1 yields
reader 2 finishes
reader 3 finishes
reader 1 finishes
reader 4 finishes
writer 3 writes: abc
writer 3 finishes
writer 4 writes: abcd
writer 4 finishes
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

Challenge3: 调研 kfifo 可不可以移植到 nachos 中

“kfifo,在 Linux 内核文件 kfifo.h 和 kfifo.c 中，定义了一个先进先出圆形缓冲区实现。如果只有一个读线程、一个写线程，二者没有共享的被修改的控制变量，那么可以证明这种情况下不需要并发控制。”【1】

之所以一般的缓冲区不能同时读写，是因为只有一个指针，任何一个线程改变这个指针都会引起另一个线程的错误。环形队列可以同时读写是因为有首指针和尾指针两个指针，不相互影响，所以可以不需要别的同步机制。

Kfifo 可以移植到 nachos 中，把首指针和尾指针定义为无符号整型，这两个指针一直单调递增即可。利用取余操作就可以得到读写操作在缓冲区中的真实位置。判断写指针是否大于读指针加缓冲区长度，就可以避免缓冲区溢出。

内容三：遇到的困难以及解决方法

困难 1：在实现生产者消费者问题时对锁的顺序没想清楚

我最初把互斥信号量 Mutex 包在最外面，里面是对队列 Full 和 Empty 的 P，V 操作，但是发生了死锁的情况。经过分析，我发现当一个线程对 Full 或 Empty 进行 P 操作后，由于它在 Sleep 之前没有释放 Mutex 信号量，导致其他线程无法对 Full 或 Empty 进行 V 操作，因此造成死锁。解决办法是把 Mutex 的 P，V 操作放在最内层。

困难 2：在实现 Barrier 时对条件变量和锁的关系比较迷惑

锁是用来保护临界区，条件变量用于通知是否有资源，它们一般成对出现。但是在 Barrier 中，并不需要保护临界区，只是需要借用条件变量的 Wait 和 Broadcast 操作。由于没有锁，所以我不知道该传什么参数给 Wait。经过思考，我认为应该传空指针，并且应该在 Wait 和 Broadcast 中对 Lock 的值进行判断，支持空指针这种情况。

困难 3：在测试读写锁时难以创造并发运行的场景

需要让 read 有机会交叉运行，才能证明多个读者可以同时读。为了实现这个需求，我在读取 buffer 后让当前线程 Yield，这样其它读者就能上 CPU

困难 4：借用信号量实现 Barrier

怎么让线程到截止点的时候通知主线程？我选择使用信号量。我把信号量的初值设置为线程数的相反数加一，这样需要所有线程都执行 V 操作后，相应的 P 操作才可以运行。但是 nachos 的信号量不支持负值。为此，我修改了信号量的实现，使它可以支持负值。相比使用一个全局的计数值，使用信号量的办法封装性更好。

内容四：收获及感想

锁和条件变量的使用变化多端，它们共同作用的时候很容易死锁。在做这次 lab 时，我养成了写前先思考的习惯。写程序固然不容易，但怎么证明自己的程序正确是更加困难的，我需要为程序创造多种复杂场景，并且在合适的位置打印中间结果。这个过程让我对程序的理解更深了。

内容五：对课程的意见和建议

我认为可以给每次的 exercise 和 challenge 加上一个难度星级评分，方便大家参考，做出来难题也可以有更多的奖励。

另外，助教可以在课堂上总结一下大家的完成情况。

内容六：参考文献

【1】linux 内核数据结构之 kfifo　　网址：http://www.cnblogs.com/Anker/p/3481373.html
【2】linux 同步机制　　网址：http://blog.csdn.net/wealoong/article/details/7957385
【3】现代操作系统　　陈向群