目录

内容一：总体概述

本次实验主要是阅读 nachos 操作系统源代码，调研进程间的同步互斥机制。实现进程的同步互斥的基本机制，并且用这些基本的机制解决典型的同步互斥问题。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Challenge
	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

Linux 中实现的同步机制包括

- 每 CPU 变量
- 原子操作
- 内存屏障
- 自旋锁
- 顺序锁
- 读-拷贝-更新（RCU）
- 信号量
- 禁止本地中断

每 CPU 变量，指的是 linux 为每个 CPU 都分配了自己的变量，各个 CPU 在运行的时候只访问自己的 CPU 变量，这样就能让各个 CPU 需要的变量能够被互斥访问。

原子操作。Linux 提供了一些手段来实现操作的原子性，操作码前缀是 lock 的汇编指令，lock 会锁定内存总线，保证执行汇编指令的时候没有其他 CPU 同时读写内存。Linux 利用这个机制，封装了一系列的原子操作，用于同步。

内存屏障。编译器会将代码进行重排，内存屏障的主要作用是强制在插入的地方不进行指令重排。即屏障之前的指令一定在屏障之后的指令执行前执行。

自旋锁。自旋锁会锁住一块临界区，在进程进入临界区之前，会尝试获取自旋锁，如果当前的锁被其他进程占有，那么自旋锁将会一直循环问询，等待下去。读写自旋锁则是允许多个进程同时对一个数据结构进行读操作，但是不允许多个进程对它进行修改。

顺序锁。顺序锁和读写锁类似，但是顺序锁给了写者优先权，写者可以优先对临界区进行修改。这样读者为了保持读取数据的一致性，需要不断重复读取缓冲区中的内容。

RCU（读-拷贝-更新）RCU 机制让读者读的时候可以不加任何锁，直接读取需要读取位置的信息，而写者在写的时候需要把数据拷贝一份副本，对副本进行修改，再在适当的将修

改的内容写回去。这样就提高了读写的效率。

禁止本地中断。禁止本地中断常常和自旋锁一起使用，因为 linux 支持多处理机。在多处理机的情况下，禁止一个处理机的中断并不能保证内存中的数据不被修改。故而两者配合使用，保证操作的原子性。

信号量。自旋锁在无法获得锁的情况下会一直自旋，通过自旋，不断轮询，直到可以获得锁。这样就会大量占用 CPU 资源，信号量机制会在进程条件无法满足的情况下，主动让出 CPU，让自己等待在进程等待队列上。等到条件满足的时候，就可以被唤醒执行。

Exercise2 源代码阅读

Nachos 现有的同步机制只有 PV 操作。Nachos 实现了 PV 操作，同时定义了锁和条件变量的接口。在 synchlist.h 中定义了同步队列，同步队列定义了一个可以进行线程同步互斥的队列。具体来说。

Semaphore 类定义的是一个信号量，该类中定义了一个等待队列，一个信号量 value，和相应的 PV 操作维护这两个成员变量。

P 操作循环判断当前的信号量的值是否满足条件，如果当前信号量的值等于 0，那么该线程会陷入睡眠状态，等待在该信号量维护的等待队列上，在等待队列上的线程会在信号量的值满足条件的时候被 V 操作重新唤醒为就绪态。P 操作全程关闭中断，保证操作的原子性。

V 操作会改变信号量的值，让其满足其他进程的唤醒条件。在执行 V 操作的时候，它会将一个等待在等待队列中的线程移出，将该线程的状态设置为就绪态，挂在就绪队列上，等待下次调度机直接把该线程调度到 CPU 上。这一系列操作也会关闭中断，保证操作的原子性。

Exercise3 实现锁和条件变量

锁的实现

实现锁主要用了信号量来进行线程的同步互斥。定义一个类描述锁，该类主要的成员变量包括一个信号量，一个线程指针。信号量用来同步互斥访问相应资源，线程指针主要用来指示当前锁被哪个线程占有。维护相应变量的函数主要有 Acquire，Release。另外还有 isHeldByCurrentThread()来判断当前的线程是不是锁的持有者。

Acquire()方法用来获取锁的所有权，即尝试将该锁实例中的线程指针指向当前线程。在实现中，必须要维护这个指针的互斥访问。故而定义信号量来实现该指针的互斥访问。在尝试将该指针指向自己时，进行 P 操作，如果当前的锁已经被其他的线程所占有，那么，该线程会等待在信号量的等待队列中，通过这种方式，让线程互斥地访问该类实例中的指针变量，保证一次只有一个线程可以判断和操作，持有实例中的指针变量。

Release()方法主要用于将锁释放。在释放之前，该方法会检查当前线程是不是锁的持有者，如果不是锁的持有者，那么释放这个操作就不会有任何意义，故而，判断不是后，该方法会做一个空操作。释放操作主要是先将实例中的指针置为空，指针置为空后，对实例中的信号量进行 V 操作，让等待在该信号量上的线程可以有机会来获得锁的所有权。

锁机制可以实现对临界区资源的互斥访问。

条件变量的实现

定义条件变量的类，该类主要定义了成员变量 waitinglist，这是一个等待队列，对于任何不满足条件的线程，条件变量都会让他等待在这个等待队列上，等待下次条件满足的时候，调度上 CPU。定义了方法 wait(),signal(),broadcast()来实现对这个队列的维护。一般来说条件变量会配合锁来使用，配合锁使用条件变量的时候，需要对判断条件操作进行加锁操作。使用条件变量发生条件不满足的时候，wait 方法会将当前的线程挂在等待队列上，挂在等待队列上的线程会在条件满足的时候被唤醒。在条件满足的时候，signal 方法会唤醒等在等待队

列中的一个线程，将该线程的状态置为就绪态，等待下一次调度再次上 CPU。同时，条件变量还定义了 broadcast 方法，用来唤醒所有的等待在等待队列上的线程。Broadcast 方法主要是判断等待队列是否还有线程，有的话，直接唤醒所有正在等待的线程。

Exercise4 实现同步互斥实例

信号量实现哲学家就餐问题

定义哲学家就餐问题的模型。定义 4 个哲学家和 4 只筷子，每个筷子都是互斥变量，故而定义成互斥的信号量。规定每个哲学家在需要就餐的时候都会先取左边的筷子。可以看见的是，当所有的哲学家都需要就餐并且拿起了左手边筷子的时候，系统发生了死锁。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos
Successfully Add Thread main to GlobalList!
Successfully Add Thread Philosopher0 to GlobalList!
Successfully Add Thread Philosopher1 to GlobalList!
Successfully Add Thread Philosopher2 to GlobalList!
Successfully Add Thread Philosopher3 to GlobalList!
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher1 Want to Get Chopsticks0
Philosopher1 Get Chopsticks0
Philosopher2 Want to Get Chopsticks1
Philosopher2 Get Chopsticks1
Philosopher3 Want to Get Chopsticks2
Philosopher3 Get Chopsticks2
Philosopher0 Want to Get Chopsticks0
Philosopher0 Go to sleep
Philosopher1 Want to Get Chopsticks1
Philosopher1 Go to sleep
Philosopher2 Want to Get Chopsticks2
Philosopher2 Go to sleep
Philosopher3 Want to Get Chopsticks3
Philosopher3 Go to sleep
```

可以看见的是，由于占有资源的哲学家陷入了睡眠，导致资源无法被释放，故而等待该资源的哲学家线程也直接陷入睡眠，到最后，系统中所有的哲学家线程都无法进行。

解决方法：

定义 Server 函数，server 函数的主要功能是帮助哲学家线程分配对应的筷子资源。用 server 函数每次请求一双筷子，也用 server 函数每次归还一双筷子。这就需要保证 server 函数每次的执行不能被中断。故而再次定义一个信号量，用于互斥进入 server 函数。这样就保证了每次每个哲学家请求筷子的时候得到的都是一双而不会被打扰。

改进了之后的运行效果。如下图所示。由于不让程序无限循环下去，规定每个哲学家吃饭思考十次以后就自动结束现阶段的任务。

```
Successfully Add Thread main to GlobalList!
Successfully Add Thread Philosopher0 to GlobalList!
Successfully Add Thread Philosopher1 to GlobalList!
Successfully Add Thread Philosopher2 to GlobalList!
Successfully Add Thread Philosopher3 to GlobalList!
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher1 Go to sleep
Philosopher2 Go to sleep
Philosopher3 Go to sleep
Philosopher0 Want to Get Chopsticks0
Philosopher0 Get Chopsticks0
Philosopher0 Having A Meal
Philosopher0 Return Chopsticks3
Philosopher1 Go to sleep
Philosopher0 Return Chopsticks0
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher2 Go to sleep
Philosopher0 Want to Get Chopsticks0
Philosopher0 Get Chopsticks0
Philosopher0 Having A Meal
Philosopher0 Return Chopsticks3
Philosopher3 Go to sleep
Philosopher0 Return Chopsticks0
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher1 Go to sleep
Philosopher0 Want to Get Chopsticks0
Philosopher0 Get Chopsticks0
Philosopher0 Having A Meal
Philosopher0 Return Chopsticks3
Philosopher2 Go to sleep
Philosopher0 Return Chopsticks0
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher3 Go to sleep
Philosopher0 Want to Get Chopsticks0
Philosopher0 Get Chopsticks0
Philosopher0 Having A Meal
Philosopher0 Return Chopsticks3
Philosopher1 Go to sleep
Philosopher0 Return Chopsticks0
Philosopher0 Want to Get Chopsticks3
Philosopher0 Get Chopsticks3
Philosopher2 Go to sleep
Philosopher0 Want to Get Chopsticks0
```

条件变量实现生产者消费者问题

条件变量配合锁一同实现生产者消费者问题。定义 itemnum，当生产者生产的数量达到 2 的时候就停止生产，并且通知消费者来消费。当消费者把 item 消费完毕后，会再次唤醒生产者。如此往复 5 个循环结束。在涉及 itemnum 的地方加上锁用以互斥访问。同时在判断该变量值的时候配合条件变量使用，实现两个消费者和两个生产者之间的同步。

```
Consumer2 Go to sleep
Producer1 produce an item. Itemnum:2
Producer1 Go to sleep
Consumer1 consume an item. Itemnum:1
Consumer1 Go to sleep
Producer2 produce an item. Itemnum:2
Producer2 Go to sleep
Consumer2 consume an item. Itemnum:1
Consumer2 Go to sleep
Producer1 produce an item. Itemnum:2
Producer1 Go to sleep
Consumer1 consume an item. Itemnum:1
Consumer1 Go to sleep
Producer2 produce an item. Itemnum:2
Producer2 Go to sleep
Consumer2 consume an item. Itemnum:1
Consumer2 Go to sleep
Producer1 produce an item. Itemnum:2
Producer1 Go to sleep
Consumer1 consume an item. Itemnum:1
Consumer1 Go to sleep
Producer2 produce an item. Itemnum:2
Producer2 Go to sleep
Consumer2 consume an item. Itemnum:1
Consumer2 Go to sleep
Producer1 produce an item. Itemnum:2
Successfully Remove Thread Producer1 from GlobalList!
Consumer1 consume an item. Itemnum:1
Consumer1 Go to sleep
Producer2 produce an item. Itemnum:2
Successfully Remove Thread Producer2 from GlobalList!
Consumer2 consume an item. Itemnum:1
Successfully Remove Thread Consumer2 from GlobalList!
Consumer1 consume an item. Itemnum:0
Successfully Remove Thread Consumer1 from GlobalList!
ThreadName:main Pid:0   Uid:22  ThreadStatus:RUNNING    ThreadPriority:10
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Assuming the program memory has been recllected.
Machine halting!

Ticks: total 101300, idle 0, system 101300, 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...
```

***Challenge 实现 barrier**

Barrier 机制是 Linux 为了解决编译优化所带来问题设计的机制，这个机制设置了一个屏障，在达到屏障条件之前，屏障后的代码不会运行。定义 barrier_function 来模拟屏障机制。模拟函数主要做的事情是将自己的局部变量从 0 加到 5，当所有的线程局部变量等于 5 的时候，进行累加，赋值给全局变量。这个过程结束之后，再次进行第二次的累加。实验中开启了四个模拟函数的线程，进行两次累加。具体过程见实验结果的截图。

该屏障模拟函数主要使用条件变量配合锁进行线程之间的同步互斥。定义 flag 数组显示每个线程是否处于完成状态，这个数组就是判断条件变量的条件。每次访问和修改这个数组必须加锁。先处理好的，但是条件不满足的线程，会在进行过判断后释放锁，并且等待在条件变量的等待队列中，直到最后一个线程完成执行条件，唤醒所有线程进程累加处理。

```
Successfully Add Thread barrier_function3 to GlobalList!
current thread barrier_function3 gone sleep , next thread main is going running
current thread barrier_function2 gone sleep , next thread main is going running
current thread barrier_function1 gone sleep , next thread main is going running
Add barrier function0 countnum = 5
Add barrier function3 countnum = 10
barrier_function2 Go to sleep
current thread barrier_function2 gone sleep , next thread barrier_function1 is going running
barrier_function1 Go to sleep
current thread barrier_function1 gone sleep , next thread barrier_function0 is going running
Add barrier function2 countnum = 15
Add barrier function1 countnum = 20
barrier_function3 Go to sleep
current thread barrier_function3 gone sleep , next thread barrier_function0 is going running
barrier_function0 Go to sleep
current thread barrier_function0 gone sleep , next thread main is going running
current thread barrier_function3 gone sleep , next thread barrier_function1 is going running
current thread barrier_function0 gone sleep , next thread barrier_function3 is going running
Add barrier function3 countnum = 25
barrier_function2 Go to sleep
current thread barrier_function2 gone sleep , next thread main is going running
current thread barrier_function2 gone sleep , next thread barrier_function3 is going running
current thread barrier_function3 gone sleep , next thread barrier_function1 is going running
Successfully Remove Thread barrier_function3 from GlobalList!
Add barrier function0 countnum = 30
barrier_function2 Go to sleep
current thread barrier_function2 gone sleep , next thread barrier_function1 is going running
barrier_function1 Go to sleep
current thread barrier_function1 gone sleep , next thread main is going running
Add barrier function2 countnum = 35
current thread barrier_function0 gone sleep , next thread main is going running
Successfully Remove Thread barrier_function0 from GlobalList!
current thread barrier_function1 gone sleep , next thread barrier_function2 is going running
current thread barrier_function2 gone sleep , next thread main is going running
Successfully Remove Thread barrier_function2 from GlobalList!
Add barrier function1 countnum = 40
current thread barrier_function1 gone sleep , next thread main is going running
Successfully Remove Thread barrier_function1 from GlobalList!
ThreadName:main Pid:0   Uid:22  ThreadStatus:RUNNING    ThreadPriority:10
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Assuming the program memory has been recllected.
Machine halting!

Ticks: total 100840, idle 0, system 100840, 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
```

内容三：遇到的困难以及解决方法

困难 1

进程线程的同步互斥部分遇到的问题主要是同步互斥机制过多且繁杂。Linux 提供了玲珑满目的机制，各个机制有各个机制的运用场景和限制。一开始接触的时候未免感到力不从心。但是沉下心分析了各个机制的主要特点和实现原理，发现同步互斥机制的万变不离其宗，都是从最基本的信号量管程演变思想而来，适用于不同场景。

困难 2

对于各种问题的模型足够了解，但是实际编程实现这些模型的时候，还是需要费工夫想一想如何才能将机制展示出来。

内容四：收获及感想

同步互斥机制不管是在现在还是以后的编程中，都占有很重要的地位。多线程进程环境是现代系统的标配，如何写出正确的优雅的并发代码是编程人员必须考虑的重要的事情。这个事情稍有不慎可能带来灾难性的后果。故而，需要理解各种同步互斥机制的原理和应用场景，熟练掌握并发编程。

内容五：对课程的意见和建议

暂时无。

内容六：参考文献

[1]《nachos 中文教程》
https://wenku.baidu.com/view/905197a9e209581b6bd97f19227916888486b90b.html
[2]《nachos 学习笔记（四）》https://blog.csdn.net/darord/article/details/83303765